Delaunay Mesh Refinement on the GPU

Zhenghai Chen, Tiow-Seng Tan, and Hong-Yang Ong

Abstract—We present three GPU algorithms for performing 2D and 3D Delaunay mesh refinement. For 2D, we present a GPU constrained Delaunay algorithm which takes in a plane straight line graph as input and generates a conforming quality mesh. For 3D, we present a GPU constrained Delaunay algorithm and a GPU restricted Delaunay algorithm, which both take in a piecewise linear complex as input and generates a conforming quality mesh and an approximating quality mesh respectively. These algorithms adopt a set of design strategies, aimed at improving GPU utility, which can be possibly applied to other mesh refinement solutions. Experimental results show that our algorithms are faster than the current state-of-the-art counterparts (for both sequential and multi-threading CPU versions) by up to an order of magnitude, while maintaining similar number of Steiner points being inserted. The source codes of our mesh refinement software are made available online.

Index Terms—Parallel Meshing, Delaunay Triangulation, GPGPU, CUDA.

1 Introduction

Mesh refinement is an important computation step in many engineering and scientific applications, such as finite element analysis, interpolation, GIS, path planning, etc. These applications usually require meshes, besides being Delaunay, to conform to input constraints on line segments or facets, and to meet certain quality criteria such as bounds on the triangle area, angle, edge length, tetrahedron volume, dihedral angle, aspect ratio, etc. Quality of meshes can significantly impact the performance of applications, from the actual processing time required to even whether the applications can terminate correctly. To this end, *Delaunay mesh refinement* algorithms have been proposed [1], [2] to refine meshes by inserting additional points, called *Steiner points*, while maintaining the Delaunay property of the output mesh. It is generally not desirable for a mesh refinement algorithm to use excessively many Steiner points as a large resultant mesh generally implies that more computation is required for any following use of the mesh in applications.

There are several good mesh refinement software implementations in the past thirty years to address Delaunay mesh refinement in 2D and 3D, notably, `Triangle` by Shewchuk [3], `TetGen` by Hang [4], and `CGAL`'s 2D and 3D mesh generation packages [5], [6]. These software add Steiner points one by one (or a few at a time in multicore CPU) to refine a mesh until a quality one is reached. However, even on the most powerful CPU to date, it typically takes a very long time for such implementations to compute the result for large inputs. We thus want to explore the use of the GPU, which is relatively affordable, to do mesh refinement in a concurrent manner.

Our review of prior works shows that there was no recent notable progress in the use of GPU for Delaunay mesh refinement with input plane straight line graph in 2D or piecewise linear complex in 3D. It is known that this problem is irregular [7], as the number of required Steiner points and the associated dependency-related issues are not known beforehand. Thus, adapting the sequential approach in a straight-forward manner for a parallel version will not result in a performant algorithm.

To this end, we have studied 2D and 3D Delaunay refinement problems and published our earlier findings in two conference papers [8], [9]. Consolidating our learning, the current paper presents a set of design strategies for GPU solutions for mesh refinement problems. We show how it is used to improve the performance of our previous works [8], [9] and apply it to a different refinement problem (Section 7). Specifically, the paper makes the following contributions:

1) A set of design strategies based on changes in workload and computational resource utilization to potentially achieve better speed up on the GPU. These strategies are possibly applicable to many other mesh refinement problems.
2) A GPU algorithm for 2D constrained Delaunay mesh refinement (`gDP2d`) showing an order of magnitude speed up to the CPU algorithms in `Triangle` [3] and `CGAL` [5].
3) A GPU algorithm for 3D constrained Delaunay mesh refinement (`gQM3d`) showing an order of magnitude speed up to the CPU algorithm in `TetGen` [4]; see Figure 1(a) and (b).
4) A GPU algorithm for 3D restricted Delaunay mesh refinement (`gDP3d`) showing an order of magnitude speed up to the multi-threading CPU algorithm in `CGAL` [6]; see Figure 1(c) and (d).

Section 2 provides background on Delaunay mesh, followed by Section 3 reviewing some previous works. Section 4 describes the set of design strategies used in our proposed algorithms. Section 5, Section 6, and Section 7 present our GPU algorithms for the 2D constrained, 3D constrained and 3D restricted Delaunay mesh refinement problems respectively, with the experimental results highlighted in Section 8. Section 9 concludes the paper. Source codes, pre-built executable (for Windows) and the problem

- *Authors are with the School of Computing, National University of Singapore.*
- *Webpage: https://www.comp.nus.edu.sg/~tants/meshRefinement.html*
- *Emails: {dcschai | dcstants | dcsohy}@nus.edu.sg.*

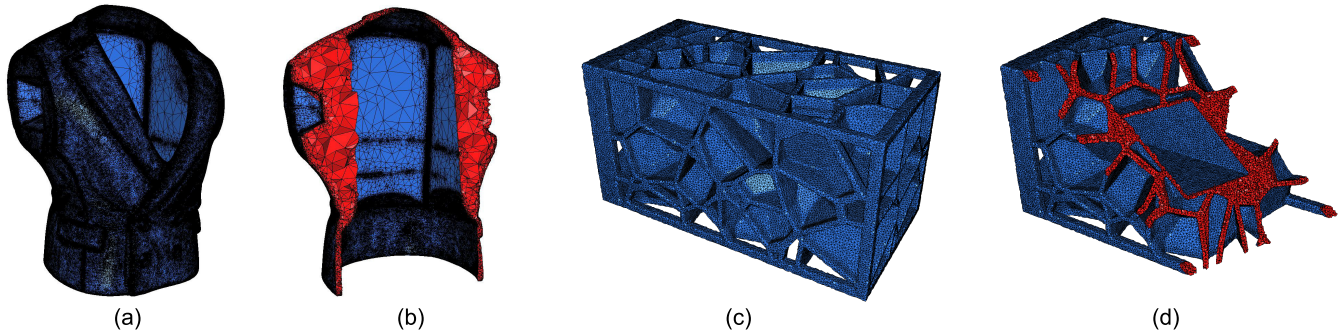


Fig. 1. Meshes generated by our GPU algorithms. (a) and (b) 3D quality CDT and its cut-off view generated by `gQM3d` on the Model 763718, Jacket. (c) and (d) 3D quality RDT and its cut-off view generated by `gDP3d` on the Model 940414, Voronoi Lamp.

datasets referred in this paper are available at our project webpage.

2 Preliminaries

In 2D, consider a *planar straight line graph* (PSLG) $\mathcal{G} = (P, E)$, where P is the set of points and E is the set of non-crossing edges (with endpoints in P). A *mesh* or *triangulation* $\mathcal{T}$ of $\mathcal{G}$ is a decomposition of the convex hull of P into triangles with all points and edges of $\mathcal{G}$ appearing as vertices and union of edges in $\mathcal{T}$. When E is empty, $\mathcal{T}$ is the (*unconstrained*) *Delaunay triangulation* of $\mathcal{G}$ if the circumcircle of each triangle t in $\mathcal{T}$ contains no other vertices. When E is not empty, $\mathcal{T}$ becomes the *constrained Delaunay triangulation* (CDT) of $\mathcal{G}$ if the circumcircle of each of its triangle t does not contain any vertices except for possibly those not visible (as blocked by edges in E) to all three vertices of t. If the mentioned exception is dropped, $\mathcal{T}$ is also a *conforming Delaunay triangulation* that conforms to $\mathcal{G}$.

In 3D, consider a piecewise linear complex (PLC) $\mathcal{G} = (P, E, F)$, where P is the set of points, E is the set of edges (with endpoints in P) and F is the set of polygons (with edges in E). Then, the Delaunay, constrained Delaunay and conforming Delaunay triangulations of $\mathcal{G}$ can be defined similar to that in 2D, but with the empty circumsphere of tetrahedron and polygons in F blocking the visibility.

For the subsequent discussion, we call an edge in E a *segment*, an edge in $\mathcal{T}$ which is also a part or whole of some segment a *subsegment*, and a triangle in $\mathcal{T}$ which is a part or whole of some polygon in F a *subface*.

There is another type of triangulation named *restricted Delaunay triangulation* (RDT) $\mathcal{T}$ of PLC $\mathcal{G} = (P, E, F)$ which *approximate* $\mathcal{G}$ for use in applications. It is a Delaunay triangulation where all vertices of $\mathcal{T}$ are sampled from edges in E and polygons in F, such that the facets (triangles) of $\mathcal{T}$ *identify* with polygons in F as follows. The *Voronoi dual* of a facet (triangle) f of $\mathcal{T}$ is either (1) a line segment joining the centers of circumspheres of the two tetrahedra (if exist) incident to f, or (2) a ray that is perpendicular to f and starts at the center of the circumsphere of the only tetrahedron incident to f. A facet f *identifies* with a polygon g in F if the Voronoi dual of f intersects g (and g is the closest to f among all such polygons in F). Such a facet f is also termed a *subface*. That is, a facet f of a RDT $\mathcal{T}$ is a subface representing the boundary of $\mathcal{G}$ when it can identify with some polygon in F; otherwise, it is in the interior of $\mathcal{T}$.

The output of a mesh refinement algorithm can be an unconstrained, constrained, conforming or restricted Delaunay triangulation, subjected to the input PSLG or PLC $\mathcal{G}$ and the user's requirements. In practice, such an output triangulation usually contains, besides the points in $\mathcal{G}$, some Steiner points inserted to improve the triangulation quality.

We adopt the triangle-based and tetrahedron-based data structure to represent the 2D and 3D triangulation respectively, both of which are shown to be effective on the CPU [4] and GPU [10], [11]. The point coordinates, vertex lists of mesh elements (vertices, subsegments, subfaces, triangles, or tetrahedra) and the neighborhood information among mesh elements are stored in simple lists, which make navigation and manipulation of the triangulation efficient. We also adopt the infinite vertex from [3] to connect all boundary edges to form virtual triangles in 2D, and to all boundary faces to form virtual tetrahedra in 3D.

3 Literature Review

The sequential and parallel algorithms for Delaunay mesh refinement are discussed in the following two subsections.

3.1 Sequential Algorithms

Given an input point set, the unconstrained Delaunay refinement produces Delaunay triangulation with well-shaped triangles or tetrahedra. Frey [13] and Weatherill [14] eliminate poorly shaped triangles from a 2D Delaunay triangulation by inserting their circumcenters and centroids respectively. Dey *et al.* [15] and Chew [16] refine a 3D Delaunay triangulation by inserting Steiner points at the centers of circumspheres of tetrahedra with bad shapes, both of which produce uniform meshes whose tetrahedra are of roughly the same size.

For 2D, Chew's algorithm in [1] performs constrained Delaunay refinement which takes a PSLG as its input and outputs a CDT, but sometimes has many more triangles in the output than necessary. To overcome this shortcoming, Ruppert's algorithm [2] and Chew's algorithm in [17] have been proposed to produce constrained and conforming Delaunay triangulation, respectively, with well-shaped triangles of graded sizes. The popular open source program `Triangle` by Shewchuk [3] and `CGAL` 2D mesh generation package [5] are widely used for constrained and conforming Delaunay refinement.

For 3D, conforming Delaunay refinement [18] produces conforming Delaunay triangulation, where each subface has

No.	Strategies	Bandwidth	Efficiency	Load Imbalance	CPU Leveraging	Load Fluctuation
S1	Compaction	X				
S2	Regularization		X	X	X	
S3	Filtering		X			
S4	Irregularization		X			X
S5	Deferment		X	X	X	

TABLE 1
Performance issues addressed by the design strategies used in mesh refinement. **S1** and **S2** are adapted from Table 2 in [12], while **S3** to **S5** are conceptualized for mesh refinements to address different issues, including the added column of load fluctuation. The other strategies of tiling, privatization, scatter to gather conversion, binning, data layout transformation and granularity coarsening in [12] are not in focus in this paper.

empty diametral sphere, by repeatedly splitting subsegments and subfaces until all segments and polygons in the input PLC appear as unions of subsegments and subfaces. However, this process may result in inserting impractically many Steiner points. By contrast, constrained Delaunay refinement [19] reduces the expected number of Steiner points needed by starting with a computed CDT of the PLC before proceeding to refining the subsegments, subfaces and tetrahedra iteratively. The popular `TetGen` [4] has a well-implemented constrained Delaunay refinement algorithm.

For restricted Delaunay refinement such as [20] and [21], the Delaunay triangulation is created from sampled vertices of the input constraints. It has several advantages over constrained and conforming Delaunay refinement: it is not dependent on the quality of the input constraints; it may lead to a better approximation of the input constraints and a better quality mesh as well. The `DelPSC` [22] and `CGAL` 3D mesh generation [6] software are based on this approach.

3.2 Parallel Algorithms

Many parallel algorithms, such as [23], [24], use domain partitioning strategy to select a set of independent points to refine the mesh. Blandford *et al.* [25] and Foteinos and Chrisochoides [26] each presents an algorithm based on memory locks. The `CGAL` 3D mesh generation software mentioned in the previous subsection also implements a CPU multi-threading version using this approach. These works incorporate one or two design strategies, but may still have room for improvement to achieve better speed up.

The works of [7], [27], and [28] build up a *pattern language* of strategies and the accompanying software framework to aid the development of parallel solutions to irregular problems such as Delaunay mesh refinement. They provide user-friendly APIs to exploit the parallelism in a wide range of irregular algorithms. By comparison, our work in this paper is of a different perspective to deal with the mesh refinement problem directly to explore all possible parallelism opportunities.

In this respect, our work is most related in perspective to [12]. They present a set of optimization patterns that can address performance issues on GPU at low and high level computation for both regular and irregular problems. We adapt some of these design strategies but also conceptualize new ones to address, besides their listed performance issues, the issue of fluctuation workload throughout the mesh refinement process. Our work focuses on high level algorithmic design strategies, as summarized in Table 1.

4 Design Strategies on Mesh Refinement

Delaunay mesh refinement is a *don't-care non-determinism* algorithm [29], i.e., different orders of refining mesh elements lead to different resultant meshes that are all acceptable. Among these results, those smaller in size are generally more desirable as inputs to downstream applications. As Delaunay mesh refinement is a process that iteratively refines *bad elements* that do not satisfy some quality criteria, it is desirable to be able to control the resultant mesh size and to try to improve the performance of every iteration.

Each iteration of Delaunay mesh refinement inserts a set of Delaunay independent points into the mesh. Two points are termed *Delaunay independent* if they are not connected to each other after their concurrent insertion into the mesh; otherwise, they are *Delaunay dependent*. The concurrent insertion of Delaunay dependent points is not allowed because they can form short edges that can no longer guarantee the termination of the algorithm. Consequently, the number of Delaunay independent points that can be inserted in one iteration has an upper bound, and will usually be much smaller than the number of bad elements considered in that iteration.

GPU algorithm consists of a serial of kernel programs, hereafter simplified as *kernels*. A kernel can launch many threads at once, where each thread can be assigned to perform some computation with respect to some unit of *work* in parallel with other threads. For simplicity's sake, our discussion assumes all work assigned to threads in a launch must be completed before another kernel can launch new threads. The duration from the time when the threads are launched to the time the slowest thread finished its work is the *latency* for the launch. And the total amount of work done for the launch is the *concurrency*. From Little's Law [30], the *throughput* for the launch is the ratio of concurrency to the latency. The design strategies aim to achieve good throughput (that, in general, also means overall good performance) for kernels used in each iteration to insert Steiner points. This can be achieved by either increasing the concurrency (subjected to the mentioned upper bound) or reducing the latency for each kernel. The following subsections discuss the design strategies presented in Table 1 with respect to scenarios of series of low workload, series of high workload, or a series of mixture of both low and high workloads. In the context of our algorithms, *workload* is determined by the number of points considered for potential insertion into the mesh as Steiner points, called *splitting points*.

4.1 Compaction and Filtering

Compaction is used to maintain potentially useful works in contiguous global memory to make it more efficient when launching processing threads. Concurrency is expected to be increased at the cost of increasing the latency. Thus, throughput can be improved if the former can well offset the latter, which is most likely when workload is high, i.e.,

there is more work than threads that can be assigned to in an iteration. On the other hand, when workload is low, i.e., there is not enough work to be assigned to most of the threads, it may be better to forgo compaction to avoid potentially decreasing the original throughput.

Filtering is a process done before compaction to mark out work which are known to be useless, so that they can be prevented from being assigned to threads subsequently. For example, when a splitting point is to be inserted as a Steiner point, no other splitting points which are Delaunay dependent can be inserted. Filtering can help prevent processing of such splitting points which will be useless. Together with compaction, filtering can reduce the expected amount of work which will end up not being useful in an iteration, but at the cost of increased latency due to its processing.

Specifically, let the ideal concurrency of the GPU be C and the effective concurrency after discounting the useless work be αC where $0 < \alpha < 1$. Then the throughput is $T = \frac{\alpha C}{L}$ by Little's Law where L is the latency. Suppose that we perform filtering before assigning the work. We improve the effective concurrency to βC, where $\alpha < \beta \leq 1$, but increased latency from L to $L + \ell$ where ℓ is the additional computational time due to the filtering (and the subsequent compaction). Then, the throughput with filtering is $T' = \frac{\beta C}{L+\ell}$. For $T' > T$, we require $\frac{L+\ell}{L} < \frac{\beta}{\alpha}$. That is, the proportion of increase in the time to do filtering must be lower than the proportion of increase in the effective concurrency. This can be used as an indicator for comparing the effectiveness of filtering methods in empirical studies.

4.2 Regularization and Irregularization

Regularization is a way to break down a work into small tokens for threads to complete with roughly the same latency. This can help to avoid unnecessarily high latency caused by large variance in the time threads take to complete work, resulting in better overall GPU utilization. In our context, the process of identifying tetrahedra in the neighborhood of a splitting points (within a mesh refinement iteration) is broken down to one thread identifying one tetrahedron at a time over many runs of a kernel in a regularized manner, instead of one thread identifying all (of some unknown number of) the tetrahedra in the neighborhood of a splitting point with just one run of a kernel.

From certain aspects, mesh refinement has to be done in groups with a certain order, prioritizing processing of some mesh elements over others by their types (e.g., edges before triangles, triangles before tetrahedra) to ensure completion in finite time and with a finite number of Steiner points. Regularization may cause work to be fragmented in such a way that results in workloads that vary greatly, for example processing only a few edges in one run resulting in low GPU utilization, to a large number of tetrahedra in another in good GPU utilization. To address this issue, we explore the merging of two or more compatible groups of previously regularized works, along with their associated kernels, into an integrated larger group to try to achieve high overall throughput. We call this design strategy *irregularization*.

Formally, let L_i and C_i be the latency and concurrency for work in group i, and L_j and C_j be those respectively for group j. Assume we merge the work in the two groups so that they are processed concurrently, with resultant concurrency C_m and latency L_m. It is expected that $C_m \geq \max(C_i, C_j)$ and $L_m = \max(L_i, L_j) + \ell$, for some value $\ell \geq 0$. Compare the expected throughput where we process the two groups concurrently $T_m = \frac{C_m}{L_m}$, against the throughput if we process the groups in series $T_s = \frac{C_i + C_j}{L_i + L_j}$. In the optimal case where $C_m \approx C_i + C_j$, we can achieve better throughput, i.e., $T_m > T_s$, if $\ell < \min(L_i, L_j)$. In the less than optimal case, however, ℓ may have to be smaller to achieve improvement, or there may even be cases where improvements are not possible due to low C_m or high ℓ. This understanding can guide the appropriate use of irregularization in practice.

4.3 Deferment

To do computation on the GPU, we usually need additional work to move and prepare the data. When these setup costs are known to be high with respect to the speed up gained from doing the computation on the GPU, it may be better to perform the computation directly on the CPU instead. For mesh refinement on the GPU, periods of low workload can occur when the computation is just starting, while the computation is in mid-progress, and when the computation is finishing. Computations that will cause low workload while starting can potentially be avoided by performing them on the CPU instead, before continuing on the GPU. Likewise, when there is low workload while the computation is finishing, it may be more efficient to transfer the current results back and finish the rest of the required computations on the CPU. However, using CPU for low workloads that happen during mid-progress is usually not feasible, due to the high cost of transferring data back and forth. Besides irregularization (Section 4.2) that can deal with low workloads, we discuss below *deferment strategy* to address two other situations of low workloads.

Within a mesh refinement iteration, the same kernel is generally launched multiple times to process each splitting point, to finally determine if it should be inserted into the triangulation. However, due to the different number of kernel launches required by each one, we may reach a case where the majority have completed their computations, while the kernel is still being launched repeatedly to process the few remaining points, resulting in increasing expected latency along with decreasing expected occupancy. One possible workaround is to stop the current refinement iteration when the number of splitting points with incomplete computation is low, and process them in the next iteration instead. This deferment can help improve the effective throughput of the current iteration by preventing the latency from getting too high while the occupancy is low, at the cost of increasing the workload of the subsequent iteration, and can be effective if thresholds are chosen such that the expected benefits outweigh the expected penalties. As a further improvement, the algorithm can be re-designed such that useful partial results from interrupted computations of the splitting points in the current iteration can be saved and re-used in the next iteration when the same splitting points are processed.

A very similar situation can occur at another level, where the instructions within a kernel can contain conditional loops. Different threads may need different number of loops

to complete, and when a small number of threads requires many more loops than the other threads, it can result in the kernel launch taking a long time to finish, even when majority of the threads are already idling. Thus, a similar approach may work: by terminating the remaining threads when majority of threads have completed, and reassigning the affected work to be processed in subsequent kernel launches. Let L be latency, and C be the effective concurrency for the original case when all completed threads wait until the slowest thread completes. Suppose that we now terminate the remaining threads after the fraction of the remaining threads reaches γ where $0 < \gamma < 1$. Let the new latency be $(L - \ell)$, where ℓ is the reduction in latency. Assuming that the terminated threads are evenly distributed with respect to effective concurrency, the mean throughput will be $\frac{(1-\gamma)C}{L-\ell}$. For the new case to be an improvement, we need $\frac{(1-\gamma)C}{L-\ell} > \frac{C}{L}$, or when simplified, $\frac{\ell}{L} > \gamma$. That is, the proportion of reduction in latency must be larger than the proportion of the loss in effective concurrency.

5 2D Constrained Delaunay

Given an input PSLG $\mathcal{G}$, angle θ and edge length l, the goal is to output a CDT $\mathcal{T}$ of $\mathcal{G}$ that contains few, if not zero, *bad* triangles. A triangle is said to be *bad* if it has an angle smaller than θ or an edge longer than l. If there are some input angles, formed by segments in $\mathcal{G}$, smaller than $60°$, the algorithm will end up with some bad triangles near the small input angles [31]. Such bad triangles are unavoidable and are acceptable in the output [2].

5.1 Overview of `gDP2d`

Algorithm 1 shows the high-level flow of `gDP2d`. It can be configured to compute CDT like Ruppert's algorithm [2], or conforming Delaunay triangulation like Chew's algorithm [17] using different definitions of encroachment: for the former, a subsegment ℓ is *encroached* if there exists a vertex p of $\mathcal{T}$ or a splitting point inside the diametric circle; it is similar for the latter with the exception of using diametric lens instead of circle. In either case, we say p *encroaches* ℓ. Midpoints and circumcenters are the splitting points associated with encroached subsegments and bad triangles respectively. Starting with a CDT $\mathcal{T}$ at `Line 1` (easily obtained using [3] on CPU or [10] on GPU), we concurrently process all mesh elements of encroached subsegments and bad triangles together (**S4**) in one iteration (`Line 2` to `Line 9`) until no further processing needed to improve the quality of the triangulation. When the circumcenters become Steiner points, they are also termed *free points* in $\mathcal{T}$.

Encroached subsegments and bad triangles that need to be processed are collected into an array L in GPU global memory (`Line 3`). The associated splitting point for each element is computed concurrently in a thread (`Line 5`). When there are few mesh elements in L to be processed, compaction (**S1**) at `Line 3` can be skipped to avoid introducing additional latency. When selecting a splitting point for insertion, there is an order of preference (`Line 6`) where the midpoint of a longer subsegment is preferred over that of a shorter one, circumcenter of a larger triangle over that of a smaller one, and any midpoint over any circumcenter.

Algorithm 1: `gDP2d`

Input: PSLG $\mathcal{G}$; constant θ and l
Output: Quality Mesh $\mathcal{T}$, which is a constrained or conforming Delaunay triangulation

```
1 Compute the CDT T of G
2 repeat
3     Collect encroached subsegments and bad
        triangles into L (S1, S2, S4)
4     if L ≠ ∅ then
5         Compute the splitting point set S of L
6         Prioritize and locate points of S in T (S1)
7         Filter S with cavity approximation (S3, S5)
8         Insert S into T with adapted Flip-Flop
            (S1, S2, S3, S4)
9 until L = ∅
```

In each thread processing a mesh element, the triangle of $\mathcal{T}$ containing its splitting point is located by walking from triangle to triangle (`Line 6`). Note that in the process of locating a splitting point of a bad triangle, we may pass by a triangle incident to some subsegment. In this case, we simply adopt the midpoint of the subsegment as the located splitting point and also mark the subsegment as encroached.

Let $\mathcal{S}$ be the set of splitting points for consideration for insertion into $\mathcal{T}$ via flip operations. A flip is an operation that replaces two triangles cab and acd sharing ac by the two alternative triangles abd and dbc. All insertions are to be done through the flip process while maintaining the Delaunay property of $\mathcal{T}$ (`Line 8`). However, to ensure termination of the algorithm with flips introducing new edges into $\mathcal{T}$, there are two scenarios which need to be identified and handled: first, when a circumcenter $s \in \mathcal{S}$ is found to be incident to some other splitting point $s' \in \mathcal{S}$ of a higher priority (i.e., s is Delaunay dependent to s') in the same iteration; second, when a free point s is found to encroach a subsegment (as s becomes a vertex of a new triangle incident to the subsegment). In both cases, s becomes *redundant* and has to be removed. To avoid many such costly removals of splitting or free points, filtering is added at `Line 7`. To further improve the effectiveness, adapted `Flip-Flop` from [11] is used at `Line 8` to perform both flip and flop operations appropriately in the same iteration where flop removes a redundant point along with its incident edges; see Figure 2. These are discussed in detail in the next two subsections.

5.2 Filtering

Each triangle in $\mathcal{T}$ can be possibly split by at most one splitting point at the beginning of one iteration. Thus, when a triangle of $\mathcal{T}$ encloses two or more splitting points, only the one with the highest priority should be retained (`Line 7`). This can be done quickly by one kernel where we assign one thread to each splitting point p and mark the triangle where p lies with its priority if the triangle is either unmarked, or previously marked with another splitting point of lower priority, after which those splitting points which successfully retain their markings on the triangles they lie in are retained in $\mathcal{S}$.

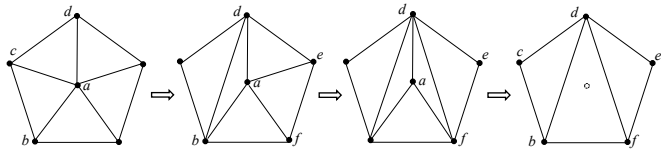


Fig. 2. A redundant point a together with its neighboring vertices are shown on the left. We first flip ac to bd, and then flip ae to df. These flips reduce the degree of a to 3. Then, we can remove a by performing the flop at a, i.e. remove the three edges ab, ad, and af together with a. For this example, it is also possible to reduce the degree of a to 3 with other flips in order to remove a. The result is a different configuration from that shown on the right.

Algorithm 2: `Flip-Flop`

Input: Splitting point set $\mathcal{S}$; CDT $\mathcal{T}$
Output: CDT $\mathcal{T}$, which is augmented with points in $\mathcal{S}$ that are not redundant points

1 Insert splitting points in $\mathcal{S}$ into $\mathcal{T}$
2 **repeat**
3 Collect flippable and floppable edges into K (**S1, S2, S4**)
4 **if** $K \neq \emptyset$ **then**
5 Prioritize and filter edges in K (**S1, S3**)
6 Perform flip and flop on K and update $\mathcal{T}$
7 Discover redundant points in $\mathcal{T}$
8 **until** $K = \emptyset$

With the above, there can still be groups in $\mathcal{S}$, each with two or more splitting points, that are Delaunay dependent to each other but not lying in a same triangle of $\mathcal{T}$ at the beginning of one iteration. Each group can only be known after all its splitting points were actually inserted into $\mathcal{T}$ and found to be connected by new edges in $\mathcal{T}$. If there are many such groups, where the splitting points will need be removed just after being added as Steiner points, it results in a lot of unnecessary work. Fortunately, it is possible to reduce this by the following filtering.

A kernel is used (`Line 7`) to assign one thread for each splitting point p to attempt to determine N_p, the set of triangles whose circumcircles enclose p, termed *cavity* of p. Starting from the triangle p lies in, for each such triangle found, its neighboring triangles will be queued for testing if it has not already been tested. However, each thread is only allowed to test up to a maximum of n triangles, where n is a predetermined integer, to prevent each from holding up the other threads which have finished their work. The incomplete work are deferred to be processed by `Flip-Flop` described in the next subsection (**S5**). Thus, the computed N_p is not the exact but approximated cavity of p. Let N_q be the computed approximated cavity of q. If N_p and N_q contain a common triangle (that can be detected via atomic operations to mark triangles), p and q are dependent and one of them is dropped from subsequent processing.

5.3 Flip-Flop

`Line 8` of Algorithm 1 is where the actual insertion of the splitting points is performed, which also removes free points and Delaunay dependent points that are discovered to be redundant along the way. It is an adaptation of `Flip-Flop` from [11], incorporating our design strategies to achieve better parallelism; see Algorithm 2. Initially, for each splitting point in $\mathcal{S}$, `Flip-Flop` uses a thread to insert it into the triangle it lies in $\mathcal{T}$, subdividing this triangle into smaller ones (`Line 1`). Then newly created edges are processed by `Line 2` to `Line 8` until the CDT property of $\mathcal{T}$ is restored and there are no redundant points, as explained below.

Subsegments in $\mathcal{T}$ are not involved in flip or flop. For an edge e in $\mathcal{T}$ which is not a subsegment, it is termed *flippable* when (i) the union of the two triangles incident to e is a convex region, and (ii) the sum of the two angles (of triangles) opposite e is larger than π or when e is connected to redundant points whose degrees are larger than 3. If e is connected to any redundant point with degree of exactly 3, it is termed *floppable*. A *flip* is performed on a flippable edge ac to replace Δcab and Δacd with Δabd and Δdbc. A *flop* is performed to remove the redundant point a as well as the three edges incident to a in $\mathcal{T}$. That is, as in Figure 2, the flop at a resulted in Δbdf replacing Δabd, Δadf, and Δabf. In general, to remove a redundant point, flips are performed to reduce its degree to 3 before a flop is performed.

Although flip and flop both use regularization (**S2**) to make workload more balanced among threads (where each thread performs a single flip or a single flop), performing flips and flops in separate kernels can result in fluctuating workload as the number of flips is usually much larger than flops, since majority of the Delaunay dependent points would already have been filtered out in `Line 7` of Algorithm 1. To address this issue, `Flip-Flop` performs flip and flop together using irregularization (**S4**). Specifically, in Algorithm 2, `Line 3` checks all edges of newly created triangles from `Line 1` or `Line 6`, and collects those that are flippable or floppable into a list K using compaction (**S1**). However, when the number of flippable plus floppable edges is small, compaction can be suppressed to avoid unnecessary increase in latency.

Before the actual flip or flop operations, edges in K are first prioritized and filtered in `Line 5` (**S3**). Floppable edges are given higher priority over flippable edges so that redundant points which have remaining degree of 3 can be removed quickly. For the same reason, among flippable edges, those connected to redundant points are given higher priority than those which are not. A flip performed on an edge e changes the two triangles sharing e, which may conflict with other flippable or floppable edges in the two triangles. Similarly, a flop affects the three triangles sharing the redundant point. To workaround this, a kernel is first executed where each thread representing an intended flip or flop operation attempts to mark the affected triangles using its assigned priority, which will only be successful if the triangles are unmarked or were marked with a lower priority. After this process, only those intended flip or flop operations which retained all their marks are executed in `Line 6`.

Flip and flop, which change the mesh by removing and creating edges and triangles, can result in more points becoming redundant. These are identified at `Line 7`: if a new edge connects one circumcenter and another splitting

point, both of which then become Delaunay dependent, the point with lower priority is marked as redundant; if a new triangle contains a subsegment that is encroached by the opposite vertex (a free point), that vertex is marked as redundant.

6 3D Constrained Delaunay

Given an input PLC $\mathcal{G}$ and a constant B, the 3D CDT problem is to compute a CDT $\mathcal{T}$ of $\mathcal{G}$ with few, if not zero, *bad* tetrahedra. A tetrahedron is *bad* if the radius-edge ratio, i.e. ratio of the radius of its circumsphere to its shortest edge, is larger than B. When boundary polygons in $\mathcal{G}$ meet at small angles, any refinement algorithm will end up with some bad tetrahedra in the vicinity of these angles [32]. In a situation where the given PLC $\mathcal{G}$ includes the water-tight boundary of an object, the refinement problem usually concerns only with the part of the triangulation enclosed within the boundary instead of the convex hull of the vertices of the PLC.

6.1 Overview of `gQM3d`

A subsegment or subface e is *encroached* when there exists a vertex p of $\mathcal{T}$ or a splitting point inside the diametric sphere of e. In this case, p is called an *encroaching point* that encroaches e. For an encroached subsegment, we use its midpoint as the splitting point; for an encroached subface, the center of its circumcircle; for a bad tetrahedron, the center of its circumsphere. Our algorithm `gQM3d` is shown in Algorithm 3. First, the CDT $\mathcal{T}$ of the input PLC $\mathcal{G}$ is computed (`Line 1`). For $\mathcal{G}$, we can assume its CDT exists or can be guaranteed by applying [33] to split some edges. This can be done directly on the CPU because it is light. Next, `gQM3d` splits the encroached subsegments with their midpoints and the encroached subfaces with the centers of their circumspheres (`Line 2`). Although this can be done on the GPU, experimentation has shown that it does not offer any particular advantage over doing it on the CPU due to requiring many iterations with light workload resulting in underutilization of the GPU. This step is thus carried out in CPU (**S5**) before moving the data to GPU to continue the rest of the computation in GPU (`Line 3` to `Line 10`).

`Line 4` first collects and irregularizes (**S4**) all encroached subsegments, subfaces and bad tetrahedra into list L, after which their splitting points are concurrently computed and stored in $\mathcal{S}$ (`Line 6`). To guarantee termination, the splitting points for mesh elements of lower dimensions are given priorities over those of higher dimensions (`Line 7`). Among elements of the same dimension, however, experimental results have shown that better speed up can be achieved by prioritizing larger elements, based on length for subsegments, area for subfaces, or volume for bad tetrahedra. This can be interpreted as trying to achieve better GPU utilization by breaking up larger elements earlier, which can create more elements to process and reduce the variance with subsequent thread latencies. `Line 7` also concurrently locates tetrahedra containing the splitting points. For $s \in \mathcal{S}$ from an encroached subsegment, it is located simply at any of the tetrahedra incident to the subsegment. For $s \in \mathcal{S}$ from an encroached subface or a bad tetrahedron, it is located by walking from tetrahedron to tetrahedron, starting from the encroached subface or bad tetrahedron.

Algorithm 3: `gQM3d`

Input: PLC $\mathcal{G}$; constant B
Output: Quality mesh $\mathcal{T}$, which is a CDT

```
1  Compute the CDT T of G
2  Split encroached subsegments and subfaces in T (S5)
3  repeat
4      Collect encroached subsegments, subfaces and bad tetrahedra into L (S1, S2, S3, S4)
5      if L ≠ ∅ then
6          Compute the splitting point set S of L
7          Prioritize and locate the points in S (S1, S3)
8          Grow and shrink cavities of points in S (S1, S2, S3)
9          Insert S into T
10 until L = ∅
```

For s to be inserted into $\mathcal{T}$ (`Line 9`), we need to compute the set of tetrahedra, called the *cavity of* s, whose circumspheres encloses s and all vertices being visible to s (`Line 8`). When s is indeed inserted into $\mathcal{T}$, the cavity of s is removed and replaced with new tetrahedra with s as a vertex, maintaining the CDT property. This dictates that no two splitting points whose cavities overlap can be inserted into $\mathcal{T}$ concurrently. Thus, during the computation of cavities (`Line 8`), some points in $\mathcal{S}$ are filtered out when their cavities are found to be overlapping with cavities from other splitting points with higher priorities. This is done by attempting to mark the tetrahedra in the cavity for each $s \in \mathcal{S}$, starting with its tetrahedron t found in the previous step (`Line 7`), in an atomic operation, where a marking is successful only if the tetrahedron is unmarked or has a marking associated with a splitting point of lower priority. All splitting points which cannot retain the markings on all the tetrahedra in their cavities will be filtered out. `Line 8` will be further detailed in the next three subsections.

At `Line 9`, the remaining splitting points can be inserted into $\mathcal{T}$ with one exception: if two splitting points s_i and s_j have cavities that share a triangle f that is not a subface, they may be Delaunay dependent and require $s_i s_j$ to be inserted to the CDT. If this edge is too short, the algorithm may not terminate. This can be avoided by performing a simple operation before any insertion: if the circumsphere of the tetrahedron formed by s_i and f encloses s_j, remove either s_i or s_j (whichever having the lower priority). After the insertion, some housekeeping has to be done to maintain correct cavity and neighborhood information, further discussed in Section 6.5.

6.2 A Tuple-based `ExpandList` Algorithm

Besides the basic triangle and tetrahedron data structure, higher-level general tuple-based lists (arrays) in the GPU global memory are used to support concurrent computations by thousands of threads using regularization (**S2**). For example, the list L in `Line 4` of Algorithm 3 is a tuple list. Given a tetrahedron t with its one face f, we write it as

Algorithm 4: `ExpandList`

Input: Tuple list H; `Predicate`; `Op_True`; `Op_False`
Output: Augmented tuple list H

```
1  Set left = 0, and right = length(H) −1
2  repeat
3      for each tuple h ∈ H[left ... right] do
4          if Predicate(h) then
5              Op_True(h)
6          else
7              Op_False(h)
8      Filter out invalid tuples in H (S1, S3)
9      Set left = right + 1, and right = length(H) −1
10 until left > right
```

a tuple $\langle t, f\rangle$. Then, the tetrahedron t' sharing the face f with t, which can be obtained in constant time from the standard tetrahedron data structure, is written as $\langle t', f\rangle$. For a mesh element e, we write $\langle e, \langle t, f\rangle\rangle$ to indicate that face f of tetrahedron t is relevant in some way to e.

To support the dynamic nature of growing or shrinking of tuple-based lists in our algorithm, we design the generic Algorithm 4 that operates on a tuple list H. In each iteration, the algorithm performs concurrent operations on a range of tuples (the *operation window*) in the list H indicated by index positions *left* and *right*. Each tuple h can be processed by one of the two pre-designated operations `Op_True` and `Op_False` depending on the result of the `Predicate` function which is first applied to it, allowing tuples that require non-similar operations to be processed together within the same iteration. Either function can modify existing tuples in H or add a pre-determined fixed number of new tuples to another tuple list or H itself (positions are reserved using prefix sum), thus allowing H and its operation window to change dynamically during processing.

In `Line 8` of Algorithm 4, compaction (**S1**) and filtering (**S3**) of tuples in H can be suppressed if the operation window does not have many tuples. In other words, we can skip the compaction and filtering when it is likely to be ineffective, or even counter-productive due to the inherent cost of increasing expected latency.

6.3 Growing Cavity

The cavities for splitting points can have a large variance in size (in number of tetrahedra). Finding the complete cavity of a splitting point in a single thread will result in severely unbalanced work: threads which completed their work very early will be forced to wait a long time for the slowest thread. To address this, we partition the work of computing a single cavity further, so that each part can be processed by a separate thread (regularization, **S2**), thus reducing the expected variance in latencies. This approach uses tuples and is fully compatible with `ExpandList` (Algorithm 4).

For a mesh element e in L (Algorithm 3) with $p \in \mathcal{S}$, T_p denotes all tetrahedra in $\mathcal{T}$ intersecting p. Initially, tuples in $\{\langle e, \langle t', f\rangle\rangle | f$ is a facet of $t \in T_p$, $f \cap p = \emptyset$ and $t \cap t' = f\}$ are added to H. That is, the initially known cavity of p consists of those tetrahedra t in T_p, and the neighbors t' of t are then added to H to be processed. Here, the `Predicate` is `insphere`, where given a $h = \langle e, \langle t', f\rangle\rangle$, returns `True` when the circumsphere of t' encloses p (retrieved via e); otherwise it returns `False`. For `Op_True`, it will attempt to mark t' with the priority of p in an atomic operation to include t' into the cavity of p. If successful, it continues to add the other neighbors (at most three excluding t) of t' to H to be processed in the next iteration; if not successful, p is to be removed from $\mathcal{S}$. As for `Op_False`, since a tetrahedron t' immediately outside the cavity of p is found, it can be added in a new tuple $\langle e, \langle t', f\rangle\rangle$ to $\overline{H}$, which keeps track of all tetrahedra that failed the `insphere` test , which will be used subsequently for the shrinking of cavities.

When the whole `ExpandList` has been processed (loop in Algorithm 4 terminates), each mesh element e with its corresponding tuples $\langle e, \langle t, f\rangle\rangle$ that successfully retain their markings on all t will have identified all the tetrahedra that may be part of the cavity of p where p is the splitting point of e. The workload is typically high when growing cavity `Line 8` (Algorithm 3) as the list L (`Line 4`) is directly dependent on the number of encroached subsegments, subfaces and bad tetrahedra collected. Thus, it is desirable to apply filtering (**S3**) to reduce waste in computation due to dependency conflicts. One such means explored is to use 3D grid approximations of the cavities of splitting points as a quick test for possible conflicts, similar to `gDP2d` for the 2D case. In practice, L is observed to be capable of becoming very large without filtering, which can result in insufficient memory on the GPU. To address this, L is kept sorted by priority and only the portion with the highest priorities is maintained on the GPU.

The above describes the case where cavities are formed by tetrahedra. As subfaces are possible constraints from input, *subface cavities*, i.e., cavities formed by triangles, may also need to be identified. Growing cavities on triangles is similar to the tetrahedra case, with the following two key differences: (1) the `insphere` test of the latter is replaced by the `incircle` test, which returns `True` if a point is inside the circumcircle of the triangle; (2) growing with tetrahedra can cross subfaces but growing with triangles must respect visibility and cannot cross subsegments.

At this point, even though some splitting points with cavities that are non-overlapping have been identified, not all can be inserted into $\mathcal{T}$ yet, as there may be other splitting points with higher priorities that are conflicting. More specifically, splitting points of bad tetrahedra that encroach existing subfaces or subsegments, and splitting points of subfaces that encroach existing subsegments cannot be performed until these encroached subfaces and subsegments have been split; see Figure 3. Disregarding occlusions due to subfaces during growing of cavity enables easy identification of splitting points that are not good candidates for insertion in the following manner. Consider a tuple $\langle e, \langle t, f\rangle\rangle$ in H corresponding to a splitting point p of e. If e is a bad tetrahedron and f is a subface, check if f is encroached by p; if e is a subface or a bad tetrahedron, and f is bounded by some subsegments, check if any such subsegment is encroached by p. If the check is `True` in either case, p is filtered out in the current round of point insertions; otherwise, if f is a subface, $\langle e, f\rangle$ is added to another list H_s which will be used (together with $\overline{H}$) for shrinking of

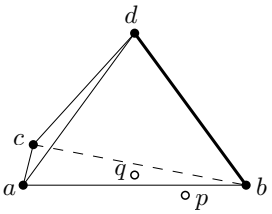


Fig. 3. An example where p is the splitting point of some bad tetrahedron with its cavity grown to include subface abc and subsegment bd. As p encroaches upon abc, it cannot be inserted before the insertion of the center q of the circumcircle of abc. In this example, q in turn encroaches upon bd and thus cannot be inserted until bd has been split into two subsegments. In another setting, it can happen that p encroaches upon bd but not necessarily abc. Because visibility was not taken into consideration in the growing of the cavity of p, all these encroachments can be discovered by checking tuples in H to avoid any unnecessary attempts to insert p or q before splitting bd.

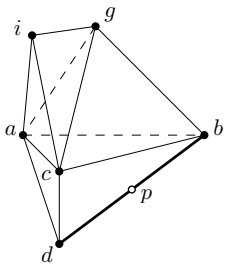


Fig. 4. An example where p is the splitting point of subsegment bd and abc is a subface in the cavity of p. Initially, $\langle bd, \langle abcg, acg\rangle\rangle$, $\langle bd, \langle abcg, abc\rangle\rangle$, $\langle bd, \langle abcg, bcg\rangle\rangle$ and $\langle bd, \langle abcg, abg\rangle\rangle$ are tuples of $\overline{H}$ as an input to `ExpandList`. For $\langle bd, \langle abcg, acg\rangle\rangle$, its neighbor is $\langle acgi, acg\rangle$ with i and p on different sides of acg. Thus, $acgi$ is marked as no longer in the cavity of p, and three new tuples $\langle bd, \langle acgi, aci\rangle\rangle$, $\langle bd, \langle acgi, cgi\rangle\rangle$, and $\langle bd, \langle acgi, agi\rangle\rangle$ are added into $\overline{H}$ for further exploration by `ExpandList`. As for $\langle bd, \langle abcg, abc\rangle\rangle$, its neighbor $\langle abcd, abc\rangle$ has vertex d and p on the same side of abc, and this neighbor thus remains in the cavity of p and is added to H_r for use in remeshing.

cavities discussed in the next subsection.

6.4 Shrinking Cavity

Since visibility was not respected during growing of cavities by tetrahedra, it needs to be addressed by a cavity shrinking process. First, for each tuple $\langle e, f\rangle$ in H_s, with p as the splitting point of e, a thread is used to first determine if f is an interior subface of the cavity of p; i.e., when both incident tetrahedra to f are marked with the priority of e. If so, the tetrehedron t with a vertex that lies on the opposite side of f with respect to p is excluded from the cavity of p as it is not visible to p. For each of the four faces f' in t, the tuple $\langle e, \langle t, f'\rangle\rangle$ is added to $\overline{H}$ (containing tetrahedra which failed the `insphere` test during cavity growing stage).

Next, $\overline{H}$ is an input to `ExpandList` for processing. Here, the `Predicate` is the `diffSide` test that takes a tuple $\langle e, \langle t, f\rangle\rangle$ due to splitting point p, and checks the vertex v (not on t) of the other tetrahedron t' (if exists) sharing f with t to return `True` when v and p are on different sides of the plane passing through f; otherwise, it returns `False`. `Op_True` marks t' to exclude it from the cavity of p, then adds one tuple $\langle e, \langle t', f'\rangle\rangle$ for each $f' \neq f$ of t' to $\overline{H}$. `Op_False` adds the tuple $\langle e, \langle t', f\rangle\rangle$ to a new list H_r, used for remeshing described in the next subsection; see Figure 4 for an illustration. When the above-mentioned process has finished, all tetrahedra known not to be in any cavity are in $\overline{H}$. As such, the list H can be updated through $\overline{H}$ to keep only tuples (of tetrahedra) of cavities, and then $\overline{H}$ is no longer needed and is set to empty for subsequent use.

At this point, one last filtering process remains that will ensure that only the splitting points which are indeed mutually Delaunay independent are retained in $\mathcal{S}$. Consider two splitting points p and q. They are Delaunay independent if the intersection of their cavities is either empty or comprises only of vertices, edges, or triangles which are subfaces. If the cavities of p and q share a triangle f which is not a subface, one of them has to be filtered out of $\mathcal{S}$ to ensure Delaunay independence among the remaining splitting points, using the following procedure: for each $\langle e, \langle t, f\rangle\rangle$ in H, if f is incident to two tetrahedra (including t) that are also in the cavities of two splitting points p (of e) and q (of the other element), if q is in the circumsphere of tetrahedron formed by p and f, then the splitting point with the lower priority between p and q is filtered out of $\mathcal{S}$. After this, the splitting points in $\mathcal{S}$ are ready to be inserted into $\mathcal{T}$.

6.5 Remeshing Cavity

To remesh cavities, the boundary tetrahedra of the cavities are derived from H_r. Recall that a tuple $\langle e, \langle t, f\rangle\rangle$ was added into H_r because it was uncertain whether t is actually inside the cavity of the splitting point of e as t may be excluded from the cavity in subsequent processing. Each tuple $\langle e, \langle t, f\rangle\rangle$ in H_r is checked and removed if t is no longer marked by the priority of e. Then, for each $\langle e, \langle t, f\rangle\rangle$ in H_r, identify each neighbor t' sharing the triangle f and add the tuple $\langle e, \langle t', f\rangle\rangle$ into $\overline{H}$ (which was set to empty for reuse here after completed its purpose in shrinking cavities).

To create the new mesh, for each tuple $\langle e, \langle t, f\rangle\rangle$ in $\overline{H}$, a new tetrahedron is created from f and the splitting point p of e. The neighborhood information is updated in two parts. First, each new tetrahedron is updated straightforwardly with (at most) one neighbor outside the cavity as available in $\overline{H}$. Second, two new tetrahedra adjacent to each other inside the same cavity need to be updated as neighbors. This is achieved for each such tetrahedron t by walking the tetrahedra outside the cavity around an edge of t (and of its neighbor) until its neighbor inside the cavity is reached.

Similarly, for each tuple $\langle e, \langle f, h\rangle\rangle$ in the list used to record subface cavity, a new subface is formed by edge h and p which is the splitting point of e. In addition, if p is on a subsegment ab, two new subsegments pa and pb are created. The neighborhood of the new subfaces and subsegments are then updated, which can be done straightforwardly.

7 3D Restricted Delaunay

Given an input PLC $\mathcal{G} = (P, E, F)$, and constants B, R, θ, r, the 3D RDT problem is to compute a RDT $\mathcal{T}$ of $\mathcal{G}$ that contains no bad subfaces and no bad tetrahedra. A subface (triangle) f is *bad* if its minimum angle is smaller than θ, or if the radius of its *Delaunay sphere*, the circumsphere of f centered at the intersection point of the Voronoi dual of f and the polygon identified with f, is larger than r; see Figure 5. A tetrahedron t is *bad* if its radius-edge ratio is

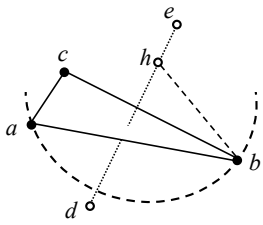


Fig. 5. a, b and c are three points sampled on E or F of the input PLC $\mathcal{G} = (P, E, F)$. These points form a facet abc in the Delaunay triangulation $\mathcal{T}$ of the set of sampled points. For here, this facet is a subface as its Voronoi dual segment de, where d and e are the centers of circumspheres of the two tetrahedra incident to abc, intersects some polygon of F at point h. So, abc is one among the many facets to approximate the boundary of F. Facet abc is bad if its minimum angle is smaller than θ or the radius $|hb|$ of its Delaunay sphere is larger than r. In this case, one or more sampled points are needed to refine $\mathcal{T}$ and thus replace abc with other subfaces.

Algorithm 5: `gDP3d`

Input: PLC $\mathcal{G} = (P, E, F)$; constants B, R, θ and r
Output: Restricted Delaunay triangulation $\mathcal{T}$ of $\mathcal{G}$ with no bad subface nor bad tetrahedra

```
1  Compute a T with a small number of vertices in P
2  repeat
3      Collect bad subfaces, bad tetrahedra into L
         (S1, S2, S3, S4)
4      if L ≠ ∅ then
5          Compute the splitting point set S of L
6          Prioritize and locate the points in S (S1, S3)
7          Compute cavities of points in S
             (S1, S2, S3, S5)
8          Insert S into T
9          Update status of facets and tetrahedra
             (S1, S2, S3, S4)
10 until L = ∅
```

larger than B, or if the radius of its circumsphere is larger than R.

To simplify the discussion, we assume that polygons in F form a water-tight 3D volume, i.e., a RDT $\mathcal{T}$ of $\mathcal{G}$ is a 3D complex formed by a collection of tetrahedra that defines the interior or volume of $\mathcal{T}$. However, it should be noted that the discussion remains applicable to a more general $\mathcal{G}$ (Section 8.3 shows some such examples in the experimentation).

An approach to the problem such as [6], [20], [21] is to iteratively refine a bad subface or a bad tetrahedron by inserting Steiner points until no bad subfaces or bad tetrahedron remains. In the case of a bad subface, the center of its Delaunay sphere (such as h in Figure 5) is used as a splitting point for potential insertion into $\mathcal{T}$ to be a Steiner point. In the case of a bad tetrahedron, the center of its circumsphere is used instead. During the insertion process, splitting points of bad subfaces are prioritized over those of bad tetrahedra. And, facets which were subfaces may lose that status when they become interior facets. Likewise, new facets may become new subfaces of $\mathcal{T}$.

7.1 Overview of `gDP3d`

`gDP3d` (Algorithm 5) works by inserting multiple Steiner points concurrently in each iteration (`Line 2` to `Line 10`) until no bad subface or bad tetrahedron remains. Before this, it first computes a small Delaunay triangulation $\mathcal{T}$ (`Line 1`) from a random sampling of vertices from P, which can be done on the CPU since it is lightweight, or directly on the GPU. To assist in the computation, $\mathcal{T}$ is placed in a bounding box large enough to contain all elements of $\mathcal{G}$. Both the interior of $\mathcal{T}$ and the exterior of $\mathcal{T}$ in the bounding box are triangulated with tetrahedra.

Algorithm 5 is similar to Algorithm 3, but uses `ExpandList` for growing cavities (but not shrinking) and intersection testing (detailed in the next subsection). Note that splitting points can have varying sizes of cavities. Although `ExpandList` is designed to try to reduce the imbalance in the work among threads when growing the cavity, it is still possible to reach a particular case when most growing are done and the few remaining ones continue to be worked on over many iterations (`Line 2` to `Line 10`, Algorithm 4). In other words, there are many iterations with low workload which underutilizes the GPU. To address this, the implementation (`Line 7`, Algorithm 5) is designed to allow deferment (**S5**) of the growing of cavities but to record the partial cavities, so that when the work is rescheduled, it can be continued from where it was interrupted. In this particular case, when the number of remaining tuples in H becomes low, the incomplete cavities are recorded so that if they are still incomplete when `ExpandList` terminates, they can be attached to their associated bad subfaces or bad tetrahedra which will become part of L (`Line 3`, Algorithm 5) to be processed in the subsequent iteration.

$\mathcal{T}$ approaches $\mathcal{G}$ as more tetrahedra are added to it. During the whole process, the algorithm needs to keep track the various status of the facets and tetrahedra, updating them as necessary. At `Line 9`, all existing subface designations are cleared, and new subfaces are identified among the facets of new tetrahedra where their Voronoi duals intersect polygons in F. In addition, new tetrahedra created at cavities are marked as either being interior or exterior to $\mathcal{T}$ which has implications: interior tetrahedra are part of the desired result and should not be bad; exterior tetrahedra have no restrictions but are not candidates for subsequent refinement, though they may be replaced in the process. This is determined by an intersection testing process detailed in the following subsections.

7.2 Intersection Testing by `ExpandList`

New facets and tetrahedra are formed at `Line 8`. Newly created facets, together with existing subfaces, have to be processed to identify the updated set of subfaces (`Line 9`): a facet is (or remains) a subface if the line segment of its Voronoi dual intersects some polygon in F. For newly created tetrahedra, they need to be categorized between being interior or exterior (`Line 9`): a tetrahedron is interior if a ray originating within it (approximated by a line segment ending at where it intersects the bounding box) intersects an odd number of polygons in F; otherwise, it is exterior.

Both the procedures on facets and tetrahedra requires intersection tests of line segments with polygons in F. To

support this on the GPU, a balanced AABB (Axis-Aligned Bounding Box) tree is constructed over all polygons in F and stored as a flat array in the GPU global memory, where each node records the bounding box over the polygons represented in its subtree. As such, the root node holds the bounding box over all polygons in F. At each node, the polygons are partitioned based on their ordering along the longest axis of the bounding box, where first half of the polygons are assigned to the left child, while the remaining are assigned to the right child, until each leaf node contains exactly one polygon from F. The AABB tree can be computed efficiently on the CPU before the result is moved to the GPU where the actual intersection tests are performed. For a given line segment r, simplified intersection tests are performed with the bounding box of a node (starting with the root node) and traversed down the children nodes if they are positive. This continues until the leaf nodes are reached, where the intersection test between r and the polygon stored in each leaf is performed. In practice, the height of the AABB tree is usually small ($\leq$ 20 in our experiments) and the intersection tests can reach the leaf nodes quickly.

With the AABB tree, `Line 9` of Algorithm 5 works as follows. For each new tetrahedron, there are 5 line segment intersection tests required: four from the Voronoi dual of the facets for the subface test, and one representing the ray for the interior/exterior test. The `ExpandList` (Algorithm 4) is implemented for performing these intersection tests concurrently on each new tetrahedra plus one each to re-classifying existing subfaces. Each line segment ℓ to be tested is stored as a tuple $\langle \ell, node \rangle$ in H to be processed by a thread, where *node* is initialized to be the root node of the AABB tree over the polygons in F. The `Predicate` returns `True` for $\langle \ell, node \rangle$ if ℓ intersects the bounding box or the polygon of F stored in the *node*; it returns `False` otherwise. For `Op_True`, if the node has child nodes, two tuples $\langle \ell, lnode \rangle$ and $\langle \ell, rnode \rangle$ are appended to H, where *lnode* and *rnode* are the left and right child nodes of *node* in the AABB tree respectively. `Op_False` marks the tuple to indicate that it no longer requires processing. When the `ExpandList` process has completed, each remaining tuple $\langle \ell, node \rangle$ in H represents the intersection of a line segment ℓ and a polygon stored in *node*. If ℓ originates from a Voronoi dual of some facet f, f is marked as a subface. If ℓ originates from a ray out of a tetrahedron t, t is interior if the number of remaining tuples in H with ℓ in them is odd. Otherwise, t is exterior.

7.3 Considerations on Intersection Testing

Regularization (**S2**) and filtering (**S3**), as considered in the intersection testing using `ExpandList`, are discussed in the next two paragraphs, followed by irregularization (**S4**) in the remaining two paragraphs.

An alternate way of performing intersection testing using the AABB tree mentioned in the previous subsection is to perform full traversals of the tree in a thread to locate and count the actual intersections. This naïve method, however, suffers from unbalanced work because the actual number of required intersection tests is unknown for each line segment. In comparison, `ExpandList` implements regularization (**S2**) by vectorizing the work into traversal steps, such that better GPU utility can be achieved.

A facet f may have some of its vertices sampled from polygons in F. In this case, the polygons used are in the vicinity of f which makes it more likely that the Voronoi dual of f intersects them. Thus, it is useful to first check for such an occurrence in a thread per facet and process it accordingly: if intersection does occurs, f will be classified as a subface, and will be omitted from the `ExpandList` processing; otherwise, a tuple associated with the Voronoi dual of f will be inserted into H for processing by `ExpandList` (described in previous subsection). The above procedure, which excludes some facets from being processed in the `ExpandList`, is a manner of filtering (**S3**).

Algorithm 5 processes the group of bad subfaces together with the group of bad tetrahedra within the same iteration (`Line 2` to `Line 10`). Although subfaces have priority over tetrahedra during processing, irregularization (**S4**) is adopted to allow concurrent processing of both element types as a single group with incorporated order dependency resolution. This is an attempt to improve the expected throughput by increasing GPU occupancy. However, it comes with the disadvantage of higher expected latency. For example, intersection testing for a subface is expected to be a lot cheaper than that for an interior/exterior tetrahedron. But after they are unified into a common work pool, they will all effectively bear the same expected cost at the slowest thread. The implementation of `gDP3d` takes into consideration the following two cases when adopting irregularization.

At the early stages in the refinement, where $\mathcal{T}$ does not yet form a good approximation of $\mathcal{G}$, high GPU occupancy caused mainly by the large number of operations on facets is expected. Adopting irregularization here can penalize the performance as the resultant latency will be increased despite minimal or even no expected improvements to the effective occupancy on the GPU. However, when the subfaces in $\mathcal{T}$ starts to approximate the boundary of $\mathcal{G}$, there are many more tetrahedra in relative comparison to a small number of subfaces to be processed to meet the quality criteria. At this stage, irregularization is expected to perform better because of the higher expected cost of intersection tests for tetrahedra. Effectively, the small number of facets are just piggybacking their intersection tests. This avoids the need of additional kernel runs with low workloads to do the tests for only the facets, which will under utilize the GPU.

8 Experimental Results

All experiments presented were conducted on a PC with an Intel i7-7700k (4 Cores/8 Threads) Processor at 4.2GHz, 32GB of DDR4 RAM and a NVIDIA GeForce GTX 1080 Ti graphics card with 11GB of video memory. `gDP2d`, `gQM3d` and `gDP3d` were implemented using the CUDA programming model [34], compiled with all optimization flags enabled. For each input, the results presented, such as timing and triangulation quality, are averages over multiple runs. Robustness is ensured by using the exact arithmetic and robust geometric predicate of Shewchuk [35]. Each predicate consists of two parts: a fast check, which uses floating point arithmetic, and an exact check, which uses floating point expansion that requires a lot more temporary memory and few threads can be executed concurrently. In practice, a fast

check is usually suffice for threads to compute the parity of a predicate; otherwise, exact check is followed up for those few needed to ascertain the parity of a predicate and then to deal with degeneracy using simulation of simplicity [36] when parity is still zero. Such a 2-stage approach is also a form of the regularization strategy (**S2**).

8.1 2D Constrained Delaunay Refinement

In general, gDP2d, implemented in accordance to the design strategies, shows 50% improvement in speed up compared to our prior work gQM [8] that did not incorporate some of the strategies, in particular, irregularization (**S4**). For comparison, we use Triangle [3] as the base reference, excluding the relatively short computation time for CDT of the input PSLG at Line 1 of Algorithm 1. The results from CGAL 2D mesh generator [5] are also included. For synthetic datasets (uniform, Gaussian, disk, and circle distribution), readers are referred to Chen [37](see Chapter 5.3.1) on the findings that gDP2d is an order or two magnitude faster than CGAL and Triangle while using comparable numbers of Steiner points.

Table 2 shows experimental results on contour maps available at http://www.ga.gov.au/ under Ruppert's mode (Chew's mode yields similar findings and is thus omitted here). To obtain the results, θ is set to 20° to guaranteed termination [2], Triangle is extended to support the edge length criterion l, and l is set to a small value for gDP2d to still able to run with the available GPU memory. Note that real world datasets often contain very small input angles which result in some unavoidable bad triangles in the output triangulations, as indicated in Table 2 (fourth column, in percentage to the total area of the triangulation). CGAL inserts fewer points in general, but over-protects vicinity with small input angles (no Steiner points are allowed in those places) which results in large bad areas that may impact the desirability of the outputs for applications. By comparison, both Triangle and gDP2d inserted enough Steiner points to achieve practically zero percentage of bad areas in the outputs. The speed up in running time of gDP2d over Triangle can reach up to 16 times (see the 8.4M sample). On the other hand, gDP2d inserts slightly more Steiner points than Triangle when the input sizes are large. Figure 6(a) and (b) show the input PSLG on the sample 3.2M and its output by gDP2d.

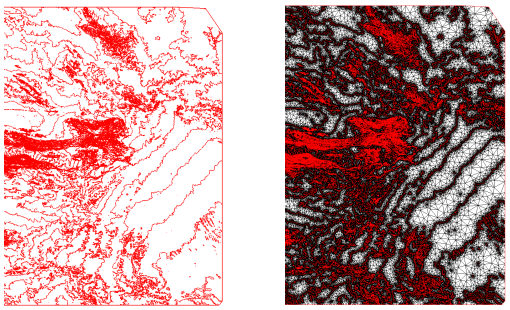
gDP2d, which incorporates all design strategies mentioned, has managed to achieve a very good overall performance. For a more in-depth analysis, variants of gDP2d are implemented and tested by removing filtering (**S3**), deferment (**S5**), and irregularization (**S4**). Removing filtering (**S3**) and deferment (**S5**) slows gDP2d down by up to 100%, while removing irregularization (**S4**) slows it down by up to 67%; see Figure 7.

8.2 3D Constrained Delaunay Refinement

gQM3d outperforms our previous work [9] by around 100%. This is mainly attributed to the implementations of the presented design strategies, made possible by the ExpandList routine. The recorded running times of gQM3d includes both the time used for actual computation and the time taken to transfer data between CPU and GPU. On synthetic datasets

File ID l	Software	Points (M)	Bad Area (%)	Time (s)	Speed Up
	Triangle	35.4	0	67	1
1.2M	CGAL	4.1	91.1	46	-
0.12	gDP2d	34.1	0	14	5
	Triangle	34.1	0	114	1
3.2M	CGAL	5.8	92.1	46	-
0.12	gDP2d	33.5	0	15	8
	Triangle	30.3	0	116	1
4.3M	CGAL	9.0	88.7	65	-
0.11	gDP2d	30.6	0	13	9
	Triangle	30.4	0	133	1
5.6M	CGAL	8.7	91.6	59	-
0.1	gDP2d	31.5	0	16	8
	Triangle	26.6	0	251	1
8.4M	CGAL	12.5	87.8	82	-
0.12	gDP2d	29.2	0	16	16
	Triangle	29.3	0	345	1
9.4M	CGAL	13.9	88.5	93	-
0.16	gDP2d	32.0	0	25	14

TABLE 2
2D CDT experimental results on some contour maps.



(a) Part of the input PSLG (b) Corresponding output

Fig. 6. gDP2d working on the sample 3.2M in Table 2.

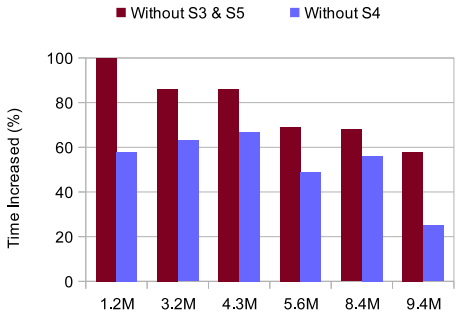
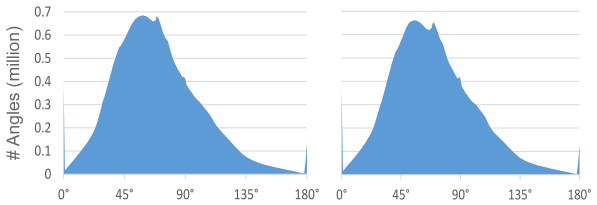


Fig. 7. Increase in running time (in percentage) of gDP2d without incorporating some of the design strategies for the real world samples in Table 2. Compaction (**S1**) is used in general where needed. It could be turned off when workload is low but we observed relatively small impact on running time. It is thus not featured in the chart. As for regularization (**S2**), it is adapted all over the algorithm and thus meaningless to turn it off to add to the chart too.

(uniform, Gaussian, ball and sphere distribution), gQM3d achieves more than 40 times speed up over TetGen [4] while inserting no more than 4% more Steiner points; details are available in [37] (see Chapter 6.2.1). gQM3d and TetGen were also tested on samples from the Thingi10k dataset [38]. Table 3 shows some samples of the results with $B = 1.4$. These samples are chosen with no particular bias other than to show the range of possibility on the speed up (sixth column). In particular, we see that gQM3d can be faster than TetGen from 9 to 101 times, while maintaining similar

Model ID Name	Software	Points (M)	Tets (M)	Time (min)	Speed Up
63788	TetGen	4.27	19.7	66.5	1
Skull	gQM3d	4.23	19.4	0.7	101
65942	TetGen	4.54	20.6	71.4	1
Sculpture	gQM3d	4.50	20.3	1.0	69
63785	TetGen	3.29	15.1	39.3	1
Half-skull	gQM3d	3.37	15.4	0.8	52
94059	TetGen	2.54	11.3	24.8	1
Mask	gQM3d	2.54	11.2	0.5	50
1717685	TetGen	2.98	13.5	24.6	1
Brain	gQM3d	2.99	13.5	0.9	28
793565	TetGen	2.13	9.6	17.2	1
SkullBox	gQM3d	2.16	9.7	0.6	28
461112	TetGen	3.29	14.9	28.9	1
Mutant	gQM3d	3.40	15.3	1.0	28
763718	TetGen	1.83	8.3	12.1	1
Jacket	gQM3d	1.79	8.1	0.5	24
252653	TetGen	2.93	13.6	29.5	1
Thunder	gQM3d	2.94	13.5	1.6	19
87688	TetGen	1.20	5.2	5.4	1
Shy-light	gQM3d	1.21	5.3	0.6	9

TABLE 3
3D CDT Experimental results on some samples in Thingi10k.



(a) TetGen (b) gQM3d

Fig. 8. The distributions of dihedral angles in the output triangulations of the Model 763718, Jacket as generated by TetGen and gQM3d.

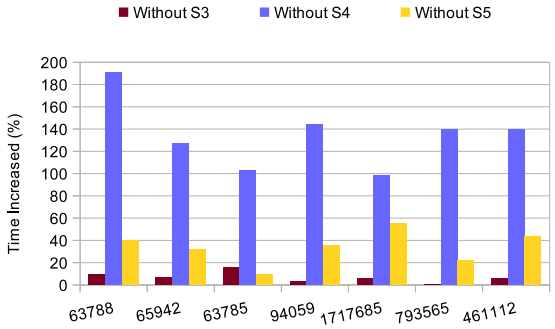


Fig. 9. Increase in running time (in percentage) of gQM3d without incorporating some of the design strategies for the first seven real world samples in Table 3. It is of the same reasoning as in Figure 7 that **S1** and **S2** are not featured in the chart.

sizes in the output triangulations (third and fourth column). Because there are already some small angles in these real-world samples, both software also generated a small percentage of bad tetrahedra due to these small angles. On the whole though, triangulations produced by both software have similar profiles with respect to the dihedral angles of the tetrahedra. For example, Figure 8 shows the dihedral angle distribution (ranging from 0° to 180°) for the output mesh of the Model 763718, Jacket shown in Figure 1(a) and (b).

Figure 9 investigates the degree of influence upon the performance by the design strategies: filtering (**S3**) used to

Model ID Name r	Software	Points (M)	Facets (M)	Time (min)	Speed Up
1706475	CGAL-S	6.60	13.2	17.5	1
Accessories	CGAL-M	6.75	13.5	45.6	-
0.08	gDP3d	6.62	13.2	0.7	25
45939	CGAL-S	5.84	11.7	12.0	1
Gandhi Litho	CGAL-M	5.98	12.0	40.8	-
0.1	gDP3d	5.86	11.7	0.7	17
236922	CGAL-S	5.89	11.8	15.4	1
Aztec	CGAL-M	6.03	12.1	13.6	1.1
0.05	gDP3d	5.91	11.8	0.6	26
551021	CGAL-S	5.35	10.7	14.5	1
Arc Triomphe	CGAL-M	5.48	11.0	10.6	1.4
0.03	gDP3d	5.37	10.7	0.6	24
65942	CGAL-S	7.35	14.7	20.5	1
Sculpture	CGAL-M	7.53	15.1	14.8	1.4
0.04	gDP3d	7.37	14.8	0.8	26
1088281	CGAL-S	8.32	16.6	449.5	1
Letter Z	CGAL-M	8.52	17.0	179.0	2.5
0.03	gDP3d	8.34	16.7	11.0	41
1255206	CGAL-S	6.42	12.8	54.4	1
Hendecahedron	CGAL-M	6.57	13.1	16.7	3.3
0.02	gDP3d	6.44	12.9	1.5	36
238423	CGAL-S	6.45	12.9	35.2	1
Treads	CGAL-M	6.61	13.2	14.7	2.4
0.04	gDP3d	6.48	13.0	1.7	20
518031	CGAL-S	6.67	13.4	30.3	1
Lampan Hack	CGAL-M	6.83	13.7	9.4	3.2
0.11	gDP3d	6.69	13.4	1.1	28
117959	CGAL-S	8.25	16.6	187.1	1
Romanesco	CGAL-M	8.43	16.9	36.5	5.1
0.05	gDP3d	8.29	16.6	5.3	35
940414	CGAL-S	5.06	10.1	84.7	1
Voronoi Lamp	CGAL-M	5.19	10.4	19.1	4.4
0.3	gDP3d	5.08	10.2	3.2	26

TABLE 4
3D RDT subface refinement results on some samples in Thingi10k.

reduce Delaunay dependency among threads when growing cavities, irregularization (**S4**) used to unify the work of splitting encroached subsegments, subfaces and bad tetrahedra, and deferment (**S5**) used to run Line 2 of Algorithm 3 on the CPU instead of GPU. The graph shows the percentage increase in running times of variants each with one of the design strategies disabled compared to the original version. It is observed that omissions of **S3**, **S4** and **S5** cause slow down of up to 15%, 190% and 55% respectively.

8.3 3D Restricted Delaunay Refinement

When comparing gDP3d to CGAL 3D mesh generators including CGAL-S for single-threading CPU and CGAL-M for multi-threading CPU, two kinds of tests were performed: subface refinement and tetrahedron refinement. For subface refinement, θ is set to 30° to guarantee termination [17] and r is set to not too small a value to ensure gDP3d can run within the GPU memory limit. For tetrahedron refinement, where the input is assumed to be water-tight, θ and B are set to 30° and 2 respectively to guarantee termination [18], and R is also set to not too small a value to ensure gDP3d can run within the GPU memory limit.

Sample experimental results on datasets from Thingi10k [38] are shown in Table 4 and Table 5. These are chosen with no particular bias from the results of larger set of test samples. The tables are to show the range of possibility

Model ID Name R	Software	Points (M)	Tets (M)	Time (min)	Speed Up
1706475	CGAL-S	7.23	44.3	12.0	1
Accessories	CGAL-M	7.60	46.5	3.8	3
0.3	gDP3d	7.32	44.9	0.7	17
551021	CGAL-S	6.33	39.3	10.4	1
Arc Triomphe	CGAL-M	6.65	41.2	1.2	9
0.13	gDP3d	6.40	39.8	0.6	17
65942	CGAL-S	7.69	46.3	13.7	1
Sculpture	CGAL-M	8.07	48.5	1.8	8
0.13	gDP3d	7.78	46.9	0.8	17
1088281	CGAL-S	6.76	39.5	80.1	1
Letter Z	CGAL-M	7.09	41.4	40.5	2
0.09	gDP3d	6.85	40.0	2.9	28
1255206	CGAL-S	6.47	40.6	11.5	1
Hendecahedron	CGAL-M	6.80	42.6	1.2	10
0.12	gDP3d	6.54	41.1	0.6	19
518031	CGAL-S	7.11	44.7	11.7	1
Lampan Hack	CGAL-M	7.47	46.9	1.1	11
0.65	gDP3d	7.19	45.2	0.6	20
940414	CGAL-S	6.43	36.4	36.3	1
Voronoi Lamp	CGAL-M	6.73	38.1	7.0	5
0.63	gDP3d	6.50	36.9	1.9	19

TABLE 5
3D RDT tetrahedron refinement results on some samples in Thingi10k. Gandhi Litho, Aztec, Treads and Romanesco in Table 4 are not water-tight with interior and thus do not appear in the above.

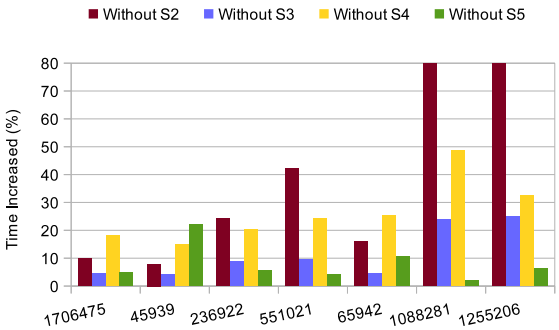


Fig. 10. Increase in running time (in percentage) of gDP3d without incorporating some of the design strategies for the first seven real world samples in Table 4. The impact of turning off compaction **S1** where applicable during low workload does not have any significant impact and thus omitted in the chart.

on the speed up (sixth column) while maintaining similar sizes in the output triangulations (third and fourth column). Specifically, for subface refinement (Table 4), the speed up of gDP3d over CGAL-S reaches up to 41 times while that of CGAL-M over CGAL-S is no more than 6 times. For tetrahedron refinement (Table 5), gDP3d can be an order of magnitude faster than CGAL-S (with up to 28 times speed up) and is a factor of at least close to 2 faster than CGAL-M. Furthermore, gDP3d inserts up to 1.3% more Steiner points compared to CGAL-S, and up to 3.8% less than CGAL-M. Besides good speed up, gDP3d produces quality output of a reasonable size. See Figure 1(c) and (d) for the quality mesh of the Model 940414, Voronoi Lamp as generated by gDP3d.

Figure 10 shows results from investigating the contribution of each of the design strategies: regularization (**S2**, using ExpandList instead of one thread per intersection test to traverse the whole AABB tree), filtering (**S3**, using a simpler preliminary check to excluding some splitting points actual tests on the AABB tree), irregularization (**S4**, does not unify intersection tests of facets and tetrahedra), and deferment (**S5**, carrying forward the computation of large cavities), by implementing and running variants of gDP3d on subface refinement each with one strategy removed. In that respective order, the slowdown observed in the variants are up to 80%, 24%, 48% and 22% respectively. Corresponding tests were performed for tetrahedron refinement and the findings were similar.

8.4 Limitations

Our implementations of gDP2d, gQM3d and gDP3d in GPU use the standard triangle-based and tetrahedron-based data structures that supported efficient walking along different mesh elements. Similar data structures are used by the CPU algorithms. However, due to additional GPU memory required for efficient 3D point insertion during the ExpandList processing, in the form of a number of auxiliary arrays, the 11GB video memory of the GTX 1080 Ti only supports the generation of 3D meshes up to 10 million points. In addition, the overheads of our GPU algorithms, especially in the cost of data transfer between the CPU and GPU, makes small problem sizes, such as those that result in outputs with only thousands of points or fewer, infeasible. It may be more practical to solve small-size problems directly on the CPU.

From experimentation, although gDP2d, gQM3d and gDP3d generate output quality meshes of sizes comparable to those of the best CPU counterparts, the output meshes can sometimes be larger though only by a few percentage. This is hard to avoid, as it is a result of optimizing efficient insertion of points (in parallel) with guarantee of termination over control of the output mesh size.

9 Concluding Remarks

This paper proposes the efficient algorithms gDP2d, gQM3d and gDP3d for 2D constrained, 3D constrained and 3D restricted Delaunay refinement problem respectively. As presented in the experimental results, they can perform better than the current state-of-the-art sequential and multi-threading CPU versions by up to an order of magnitude, while maintaining comparable sizes and good quality in the results. This can make it more feasible to integrate larger scale meshing into interactive applications in engineering and scientific domains.

The design strategies adopted have been shown to be effective for the presented problems and solutions. The simple scenarios based on expected workload or serial of workloads allows them to be easily applied or adapted to other similar general computational problems. We therefore hope that this paper is but a starting point for more future works on meshing [39] on the GPU and perhaps even other general problems beyond.

Acknowledgments

This research is supported by the National University of Singapore under grant R-252-000-678-133.

REFERENCES

[1] P. L. Chew, "Guaranteed-Quality Triangular Meshes," Department of Computer Science, Cornell University, Tech. Rep. 89–983, 1989.

[2] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation," *J. Algorithms*, vol. 18, no. 3, pp. 548–585, May 1995.

[3] J. R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in *Applied Computational Geometry Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds. Berlin, Heidelberg: Springer, 1996, pp. 203–222.

[4] H. Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator," *ACM Trans. Math. Softw.*, vol. 41, no. 2, pp. 11:1–11:36, Feb. 2015.

[5] L. Rineau, "2D Conforming Triangulations and Meshes," in *CGAL User and Reference Manual*, 5.0 ed. CGAL Editorial Board, 2019.

[6] P. Alliez, C. Jamin, L. Rineau, S. Tayeb, J. Tournois, and M. Yvinec, "3D Mesh Generation," in *CGAL User and Reference Manual*, 4.13 ed. CGAL Editorial Board, 2018.

[7] P. Monteiro, M. P. Monteiro, and K. Pingali, "Parallelizing irregular algorithms: a pattern language," in *Proceedings of the 18th Conference on Pattern Languages of Programs*, 2011, pp. 1–18.

[8] Z. Chen, M. Qi, and T.-S. Tan, "Computing Delaunay Refinement Using the GPU," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2017.

[9] Z. Chen and T.-S. Tan, "Computing Three-Dimensional Constrained Delaunay Refinement Using the GPU," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2019, pp. 409–420.

[10] M. Qi, T.-T. Cao, and T.-S. Tan, "Computing 2D Constrained Delaunay Triangulation Using the GPU," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2012, pp. 39–46.

[11] M. Gao, T.-T. Cao, and T.-S. Tan, "Flip to Regular Triangulation and Convex Hull," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 2, pp. 1056–1069, Feb. 2017.

[12] J. A. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. D. Liu, and W. Hwu, "Optimization and architecture effects on gpu computing workload performance," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[13] W. H. Frey, "Selective refinement: A new strategy for automatic node placement in graded triangular meshes," *International Journal for Numerical Methods in Engineering*, vol. 24, no. 11, pp. 2183–2200, 1987.

[14] N. Weatherill, "Delaunay triangulation in computational fluid dynamics," *Computers and Mathematics with Applications*, vol. 24, no. 5, pp. 129–150, 1992.

[15] T. K. Dey, C. L. Bajaj, and K. Sugihara, "On Good Triangulations in Three Dimensions," in *Proceedings of the First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*. New York, NY, USA: ACM, 1991, pp. 431–441.

[16] P. L. Chew, "Guaranteed-Quality Delaunay Meshing in 3D (Short Version)," in *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 1997, pp. 391–393.

[17] ——, "Guaranteed-Quality Mesh Generation for Curved Surfaces," in *Proceedings of the Ninth Annual Symposium on Computational Geometry*. ACM, 1993, pp. 274–280.

[18] J. R. Shewchuk, "Tetrahedral Mesh Generation by Delaunay Refinement," in *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 1998, pp. 86–95.

[19] H. Si and K. Gärtner, "Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations," in *Proceedings of the 14th International Meshing Roundtable*, B. W. Hanks, Ed. Berlin, Heidelberg: Springer, 2005, pp. 147–163.

[20] J.-D. Boissonnat and S. Oudot, "Provably Good Sampling and Meshing of Surfaces," *Graph. Models*, vol. 67, no. 5, pp. 405–451, Sep. 2005.

[21] S.-W. Cheng, T. K. Dey, and J. A. Levine, "A Practical Delaunay Meshing Algorithm for a Large Class of Domains*," in *Proceedings of the 16th International Meshing Roundtable*, M. L. Brewer and D. Marcum, Eds. Berlin, Heidelberg: Springer, 2008, pp. 477–494.

[22] T. K. Dey and J. A. Levine, "Delpsc: A Delaunay Mesher for Piecewise Smooth Complexes," in *Proceedings of the Twenty-Fourth Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 2008, pp. 220–221.

[23] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis, and P. Foteinos, *Towards Exascale Parallel Delaunay Mesh Generation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 319–336.

[24] C. Marot, J. Pellerin, and J. Remacle, "One Machine, One Minute, Three Billion Tetrahedra," *CoRR*, vol. abs/1805.08831, 2018.

[25] D. K. Blandford, G. E. Blelloch, and C. Kadow, "Engineering a Compact Parallel Delaunay Algorithm in 3D," in *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 2006, pp. 292–300.

[26] P. Foteinos and N. Chrisochoides, "Dynamic Parallel 3D Delaunay Triangulation," in *Proceedings of the 20th International Meshing Roundtable*, W. R. Quadros, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–20.

[27] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 65–76.

[28] R. Biswas, L. Oliker, and H. Shan, "Parallel computing strategies for irregular algorithms," *Annual review of scalable computing*, vol. 5, p. 1, 2003.

[29] E. W. Dijkstra, *A discipline of programming*. Prentice-hall Englewood Cliffs, 1976, vol. 613924118.

[30] J. D. C. Little, "A Proof for the Queuing Formula: L = λW," *Oper. Res.*, vol. 9, no. 3, pp. 383–387, Jun. 1961.

[31] J. R. Shewchuk, "Mesh Generation for Domains with Small Angles," in *Proceedings of the Sixteenth Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 2000, pp. 1–10.

[32] J. R. Shewchuk and H. Si, "Higher-Quality Tetrahedral Mesh Generation for Domains with Small Angles by Constrained Delaunay Refinement," in *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 2014, pp. 290–299.

[33] J. R. Shewchuk, "A Condition Guaranteeing the Existence of Higher-Dimensional Constrained Delaunay Triangulations," in *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, 1998, pp. 76–85.

[34] NVIDIA, *CUDA C++ Programming Guide*, 2019.

[35] J. R. Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates," *Discrete & Computational Geometry*, vol. 18, pp. 305–368, 1997.

[36] H. Edelsbrunner and E. P. Mücke, "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms," *ACM Trans. Graph.*, vol. 9, no. 1, pp. 66–104, Jan. 1990.

[37] Z. Chen, "Quality Mesh Generation on GPU," Ph.D. dissertation, School of Computing, National University of Singapore, 2020.

[38] Q. Zhou and A. Jacobson, "Thingi10K: A Dataset of 10,000 3D-Printing Models," *CoRR*, vol. abs/1605.04797, 2016.

[39] S.-W. Cheng, T. K. Dey, and J. R. Shewchuk, *Delaunay Mesh Generation*. USA: Chapman and Hall/CRC, 2012.

Zhenghai Chen obtained his PhD in 2020 at the National University of Singapore. His research interests are parallel meshing, GPGPU and geometry processing.

Tiow-Seng Tan obtained his PhD in 1992 at the University of Illinois at Urbana- Champaign. His research interests are in interactive computer graphics, GPU and computational geometry.

Hong-Yang Ong obtained his MSc in 2004, and BEng in 2001 at the School of Computing, National University of Singapore. His research interests are in GPU and computational geometry.