Computing 2D Constrained Delaunay Triangulation Using the GPU

Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan

Abstract—We propose the first GPU solution to compute the 2D constrained Delaunay triangulation (CDT) of a planar straight line graph (PSLG) consisting of points and edges. There are many existing CPU algorithms to solve the CDT problem in computational geometry, yet there has been no prior approach to solve this problem efficiently using the parallel computing power of the GPU. For the special case of the CDT problem where the PSLG consists of just points, which is simply the normal Delaunay triangulation problem, a hybrid approach using the GPU together with the CPU to partially speed up the computation has already been presented in the literature. Our work, on the other hand, accelerates the entire computation on the GPU. Our implementation using the CUDA programming model on NVIDIA GPUs is numerically robust, and runs up to an order of magnitude faster than the best sequential implementations on the CPU. This result is reflected in our experiment with both randomly generated PSLGs and real-world GIS data having millions of points and edges.

Index Terms—GPGPU, Parallel Computation, Computational Geometry, Voronoi Diagram, Image Vectorization.

1 Introduction

Delaunay triangulation (DT) is one of the most important geometric structures in computational geometry. Due to its nice property of avoiding long, skinny triangles, the DT has many practical applications in different fields. For example, in Geographical Information System (GIS), one way to model the terrain is to interpolate the data points based on the DT [13]. In path planning, the DT can be used to compute the Euclidean minimum spanning tree of a set of points, because the latter is always a subgraph of the former [23]. The DT is also often used to build quality meshes for the finite element analysis [18].

The constrained Delaunay triangulation (CDT) is a direct extension of the DT where some edges in the output are enforced beforehand [7]; these edges are referred to as *constraints*. Given a set S of n *points* (or *sites*) in the 2D plane and a set of non-crossing constraints, the CDT is a triangulation of S having all the constraints included, while being as close to the DT of S as possible. Constraints occur naturally in many applications. For example, in path planning, they are obstacles; in GIS, they are boundaries between cities; in surface reconstruction, they are contours in the slices of the body's skull; and in modeling, they are characteristic curves [3], [19], [30]. In short, the CDT complements the DT and is a very useful structure in many fields. See Figure 1 for some examples.

In another development, recently the graphics processing unit (GPU) with its enormous parallel computing power has been used widely in many disciplines, including computational geometry, for general purpose computation. Early works include computing the digital Voronoi diagram (VD) [5], [10], [17], a structure that is closely related to the DT. Recently Rong et al. [25] present a serious attempt to derive the DT from the digital VD. Their algorithm, however, is hybrid, since parallel computation is only used in the first part while leaving the rest to a sequential algorithm. As for the CDT problem, there is no efficient GPU solution as far as we know. This is partly because both the DT and the CDT problem do not present themselves readily to parallel computation. A parallel algorithm, in order to fully utilize the GPU hardware, usually needs to have regularized work and localized data access. It is not clear how to achieve those criteria while adapting the traditional and usually complex parallel techniques, such as divide-and-conquer, to both these problems.

Our main contribution here is a novel algorithm, termed *gCD*, fully parallelized on the GPU, to compute the CDT for a given PSLG. Our experiment shows that our implementation using the CUDA programming model is robust and efficient. Compared to popular software such as *Triangle* [27] and CGAL [6] as well as to the hybrid approach of Rong et al. [25], gCD runs up to an order of magnitude faster. This paper is an extension of the original version [24]. We incorporate more implementation details, complexity analysis and experiment results. Due to the page limit, the Appendices are only available as an electronic supplement document in the digital library of TVCG.

The outline of our paper is as follows. Section 2 introduces some definitions and reviews the previous works. Section 3 presents our GPU approach to the DT problem, and Section 4 extends it to compute the CDT. Experiment results and some applications are provided in Section 5. Section 6 concludes the paper.

- *All authors are with the School of Computing, National University of Singapore.*
- *Project website: http://www.comp.nus.edu.sg/~tants/cdt.html.*
- *Emails: {qimeng | caothanh | tants}@comp.nus.edu.sg.*

(a) (b)

Fig. 1. Constrained Delaunay Triangulation applications. (a) A contour map. (b) A raster image and the CDT for its edge map.

2 PRELIMINARIES

We introduce some important definitions and properties first before reviewing the related work on computing the DT and the CDT.

2.1 Terminology

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in $\mathbb{R}^2$. A *planar straight line graph* (PSLG) $\mathcal{G} = (S, E)$ is a plane graph with the point set S and the edge set E where all edges in E are straight segments.

Definition 1 (Digital Voronoi Diagram): In the 2D digital space, consider a grid of size $m \times m$, and assume all the points in S are on the grid points (which are centers of the grid cells). We say that a grid point x is *colored* by the point $p \in S$ if p is nearest to x among all points in S. In case the distances from two points p_i and p_j $(i < j)$ to x are equal, we color x by p_i. The grid with all grid points colored as above is called the *digital Voronoi diagram* (VD) of S.

Definition 2 (Delaunay Triangulation): Given a set of points S, a *triangulation* of S is a PSLG $\mathcal{T} = (S, E)$ such that $|E|$ is maximal. An edge $ab \in E$ satisfies the *empty circle property* (with respect to S) if there exists a circle passing through a and b such that no points in S are inside the circle. A triangulation $\mathcal{T}$ of S is a *Delaunay triangulation* (DT) if every edge of $\mathcal{T}$ satisfies the empty circle property.

Definition 3 (Constrained Delaunay Triangulation): Given a PSLG $\mathcal{G} = (S, E)$, two points a and b in S are *visible* from each other if the (open) line segment ab does not intersect any other edge in E. A triangulation $\mathcal{T} = (S, E')$ is a *constrained Delaunay triangulation* of $\mathcal{G}$ if $E \subseteq E'$ and each edge $ab \in E' \setminus E$ satisfies the empty circle property with respect to all the points of S that are visible from both a and b. If $E = \emptyset$, the CDT of $\mathcal{G}$ is the same as the DT of $\mathcal{G}$.

2.2 Related Work

In this subsection, we review some algorithms for constructing the DT and the CDT in 2D.

2.2.1 Delaunay Triangulation

There are many sequential algorithms developed for the CPU to compute the DT [1], [12], [29]. All these algorithms in general follow one of the three well-known paradigms: divide-and-conquer [9], sweep-line [11] and incremental insertion [15].

a. *Divide-and-Conquer*. An algorithm based on this strategy recursively divides a set of points into two smaller sets, until each set is small enough and its DT can be trivially computed. Then, the algorithm recursively merges the results of each pair of adjacent sets into that of a bigger one, until all the results are grouped to form the DT. Using this approach, the DT can be built in optimal $O(n \log n)$ time [9], [26].

b. *Sweep-line*. The VD and the DT are dual of each other. Fortune [11] uses a sweep-line algorithm to construct the VD, from which the DT is obtained. First, the algorithm sorts the input points according to their x-coordinates. Then, a vertical line, called the *sweep-line*, is swept from left to right. Points behind the sweep-line are already added into the VD, while points ahead of the sweep-line are waiting for processing. As the sweep-line progresses, the Voronoi edges are generated incrementally. The running time of this algorithm is also $O(n \log n)$.

c. *Incremental Insertion*. A natural way to compute the DT is to repeatedly insert points one at a time, re-triangulating the affected parts of the triangulation if necessary. To insert a point, we first locate the triangle containing that point. The new point splits the triangle containing itself into three triangles. Then, we perform edge flipping to obtain the DT, before inserting the next point. Though this incremental insertion approach runs in $O(n^2)$ time in the worst case, its expected time complexity is $O(n \log n)$, provided that the points are inserted in a random order [15].

2.2.2 Constrained Delaunay Triangulation

The input of the CDT problem contains both points and constraints. According to the order of handling points and constraints, the algorithms for constructing the CDT can be classified into two categories:

a. *Processing points and constraints simultaneously.* Chew [7] shows that the CDT can be built in optimal $O(n \log n)$ time using a divide-and-conquer approach.

Let us assume that all the points and constraints are contained within a given rectangle. First, we sort all the points by their x-coordinates, and use this information to divide the rectangle into vertical strips such that each strip contains only one point. In each vertical strip, we compute the CDT, followed by gluing pairs of adjacent strips together to form bigger strips, until the CDT for the entire rectangle is obtained. In a different approach, Domiter [8] uses a sweep-line to process points and constraints together. The insertion of a constraint is done once it is swept entirely, by fixing the triangles it pierced.

b. *Processing points and constraints separately.* Since the CDT is a generalization of the DT with the notion of constraints [20], we can first construct the DT of the given point set, and then insert the constraints one by one into that triangulation. Such an insertion of a constraint can be done either by removing the triangles pierced through by the constraint and re-triangulating the region due to the removal, or by flipping some edges in a certain order until the constraint appears in the triangulation [2], [27].

Our approach for constructing the CDT on the GPU lies in-between these two categories. First we construct a triangulation of the given point set, then we insert all the constraints using edge flipping, followed by transforming the resulting triangulation into the CDT, also using edge flipping.

2.3 GPU Programming Consideration

The GPU is massively multithreaded, with hundreds of processors. To effectively utilize the GPU, it is desirable to have tens of thousands of executing threads at any given time. As such, we keep in mind the following two design principles when developing algorithms for the GPU. First, the GPU architecture is most suitable for data-parallel computations, where the same computation is performed by many threads on multiple pieces of data. Therefore, we need to make our algorithm as simple and uniform as possible. That translates to writing our thread code with little complicated control flow and having similar amount of work across various threads.

Second, with so many threads, common issues in parallel programming such as cooperation among threads, conflicting data access, and racing conditions become more serious. To mitigate this, we usually employ some simple checks to break the set of tasks into several groups, within which the tasks can be done concurrently with none or fewer conflicts.

3 Computing the DT on the GPU

Our algorithm derives from the digital VD of the input point set S an approximation of the DT, then transforms it into the needed DT. The algorithm consists of the following phases:

Phase 1. **Digital Voronoi diagram construction.**
Map the input points into a grid and compute the digital VD. If several points are mapped to the same grid point, keep just one and treat the others as missing points.

Phase 2. **Triangulation construction.**
Dualize the digital VD into a triangulation by finding all the digital Voronoi vertices to form triangles. This triangulation is an approximation of the DT.

Phase 3. **Shifting.**
Points have been moved due to the mapping in Phase 1. Shift them back to their original coordinates and modify the triangulation if necessary.

Phase 4. **Missing points insertion.**
Insert all missing points into the triangulation.

Phase 5. **Edge flipping.**
Verify the empty circle property for each edge in the triangulation, performing edge flipping if necessary.

Compared to the previous hybrid approach [25], the transformation from the approximation of the DT to the DT itself (Phase 3 to Phase 5) is now completely done on the GPU. Some technicality in the approximation construction (Phase 1 and Phase 2) is also provided. We adopt the triangulation data structure used by Shewchuk [27] in our computation. A list of triangles, referred to as the *triangle list*, is stored in a pre-allocated array of size no more than $2|S|$. Each triangle has the indices of up to three other triangles edge adjacent to it. Each point in S has a linked list of triangles incident to it. These linked lists altogether form a data structure referred to as the *vertex array*. This data structure comes in handy whenever we want to visit the triangles incident to a point.

3.1 Phase 1: Digital Voronoi diagram construction

In this phase, we first translate and scale the point set such that its bounding box fits inside a 2D grid $\mathcal{Q}$ of $m \times m$ cells. Each point is mapped to the nearest lower left grid point. If several points are mapped to a same grid point, only one of them is recorded while the rest become *missing points* and will be handled later.

Second, we compute the mentioned digital VD of all points except missing points on the GPU. However, the dual of the digital VD may not be a geometrically valid triangulation, since it can have duplicate and intersecting triangles; see Figure 2. Cao et al. [4] show that by dualizing the output of the so-called Standard flooding algorithm, one gets a valid triangulation of the points. The output of the Standard flooding algorithm is very close to the digital VD except that each region of a same color is connected. We adopt the Parallel Banding Algorithm (PBA) of Cao et al. [5] to compute the digital VD on the GPU as it is much

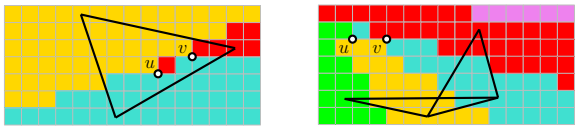


Fig. 2. (a) Duplicate and (b) intersecting triangles due to the digital Voronoi vertices u and v.

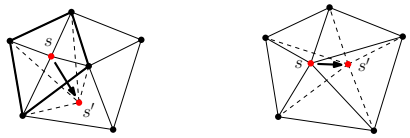


Fig. 3. Shifting a point s to s' may or may not require some modifications to the triangulation. (a) Bad case. (b) Good case.

faster than the Standard flooding algorithm. Then we quickly amend the digital VD so that the result is the same as that generated by the Standard flooding algorithm. After that, one can obtain a valid triangulation by dualizing the amended digital VD. The detailed discussion on how to amend the digital VD can be found in Appendix A.

3.2 Phase 2: Triangulation construction

In this phase, we dualize the result of the previous phase. A corner shared by up to four grid cells is incident to one to four different colors. If it is incident to three or four different colors, we call it a *digital Voronoi vertex*. For each Voronoi vertex that is incident to three colors, which correspond to three sites, we add one triangle formed by these sites into the triangulation. Similarly, for each Voronoi vertex that is incident to four colors, two triangles formed by the four corresponding sites are added.

Recall that during Phase 1, points are translated, scaled, and then slightly shifted from their original positions. We reverse this process on the constructed triangulation in two steps. First, we reverse the scaling without destroying the validity of the triangulation. Second, we shift the points and fix the triangulation if necessary. The second step is done with care in Phase 3. On the other hand, the first step, though looks trivial, can result in an invalid triangulation if the possible numerical error during the scaling process is not handled. Further discussion about this issue can be found in Appendix B.

As the digital VD is truncated within the texture, its dual is not always a complete triangulation with a convex boundary. We fix this by traversing along the boundary of the texture, using the idea similar to Graham's scan algorithm [14] to identify triangles whose Voronoi vertices fall outside the texture, and add them into the triangulation. This additional step is performed on the CPU as it is a simple task, and it can be done concurrently while the GPU is populating the triangle list.

In the later phases of our algorithm, we need to add or delete points from the triangulation in parallel. Deleting a point on the convex hull or inserting a point outside the convex hull can be quite involving, causing non-uniform parallel computations. We apply a common technique of bounding the point set with a large enough convex polygon, referred to as the *fake boundary*, to make sure all insertions and deletions are done inside the triangulation. Although a polygonal boundary with 3 points is sufficient, in our implementation we choose to use $\sqrt{k}$ points where k is the number of points on the convex hull. This helps to avoid any of them being incident to too many triangles, resulting in the work of some threads being unbalanced.

In the above computation, we want the generation of each triangle to be independent from that of other triangles in order to achieve good parallelism. Thus, each triangle is generated without linking up with the triangles sharing its edges. Once all the triangles are generated, we construct in parallel the vertex array and use that to identify for each triangle, in parallel, up to three other triangles edge adjacent to it. As a result, we obtain the data structure of a triangulation similar to that in [27].

3.3 Phase 3: Shifting

In Phase 1, we move the points during the mapping step. We should shift them back to their original coordinates. We say two points are neighbors when they are endpoints of an edge in the triangulation. Assuming the neighbors of a point s are static, shifting s may or may not cause any intersection. We refer to the former as a *bad case* and the latter as a *good case*; see Figure 3. The bad case happens when s moves across the boundary formed by its neighbors. We expect, due to the very small shifting distance, majority of the shifting in practice are good cases.

To achieve regularized work while shifting points in parallel, we separate the processing into two stages. In the first stage, we only shift points that are good cases. To do so, we perform this stage in multiple iterations. Algorithm 1 details our approach. Initially, all the points are marked as unchecked. In each iteration, each parallel thread in charge of an unchecked point s_i first verifies that all its neighbors with indices smaller than i are checked (line 4). We skip the processing of s_i if this condition is not met. Then, if shifting s_i while all its neighbors remain static does not cause any intersection, we shift it (line 6–7). Otherwise, we leave it for the second stage (line 9). After a point is processed, it is marked as checked. Since most points would be processed and marked as checked in the first few iterations, we use compaction after each

Algorithm 1 Shifting points of good cases and recording points of bad cases

```
1: set all points as unchecked;
2: repeat
3:   for each unchecked point p_i do in parallel
4:     if all neighbors p_j of p_i, j < i, have been
       checked then
5:       mark p_i as checked;
6:       if shifting p_i is a good case then
7:         update p_i to its original coordinate;
8:       else
9:         mark p_i as a bad case;
10:      end if
11:    end if
12:  end for
13: until all points have been checked;
```

iteration to skip points that are marked as checked in subsequent iterations. This helps to speed up the later computation.

In the second stage, we delete all points that are bad cases, treating them as missing points for later processing. Note that we also need the above-mentioned multiple iterations to avoid deleting in parallel two points that are neighbors; see line 2–7 of Algorithm 2. First, for each set of points to be deleted in parallel, we count their degrees. Applying parallel prefix sum on these counts, we get the starting position that each thread might use to store the indices of the triangles to be deleted (line 8). These indices are needed to store the newly created triangles during the re-triangulation. We allocate a piece of memory for each thread to store the indices mentioned above. Second, using one thread per point in parallel, we mark all triangles in its fan as deleted and store their indices in the allocated memory (line 9–11). Third, we fix the vertex array in parallel by removing the deleted triangles from it (line 12–14). After that, we use the ear-cutting method [16] to re-triangulate the resulting star-shaped holes in the triangulation in parallel, while updating the vertex array at the same time (line 15–17). Since the triangles to be created is no more than those deleted, we can use the deleted slots in the triangle list that we recorded earlier to store the new triangles, with no racing memory access during parallel computation. In the end, we update the links between the newly created triangles and those that are edge adjacent to them (line 18–20).

3.4 Phase 4: Missing points insertion

In this phase, we insert the missing points identified in Phase 1 and Phase 3 into the triangulation. The insertion of each missing point p_i starts by identifying the triangle(s), referred to as the *container*, in the triangulation that contains p_i, or has an edge passing through p_i. For each missing point obtained from

Algorithm 2 Deleting points of bad cases

```
1: repeat
2:   for each points p to be deleted as marked by
     line 9 in Algorithm 1 do in parallel
3:     if p can be processed in this iteration then
4:       Mark p as active;
5:       Record its degree;
6:     end if
7:   end for
8:   Compute parallel prefix sum of the degrees;
9:   for each p marked as active do in parallel
10:    Mark triangles in p's fan as deleted and store
       their indices;
11:  end for
12:  for each q incident to a deleted triangle do in
     parallel
13:    Fix q's linked list in the vertex array
14:  end for
15:  for each p marked as active do in parallel
16:    Triangulate the resulting star-shaped hole and
       update the vertex array;
17:  end for
18:  for each new triangle t do in parallel
19:    Compute t's 3 neighbors and update the links
       between t and its neighbors;
20:  end for
21: until all bad cases have been deleted;
```

Phase 1, we start the search from a triangle incident to the point of S that was mapped to the same grid point as p_i and was kept in the digital VD computation. As for one obtained from Phase 3, we start the search from a triangle incident to a neighbor p_j of p_i before p_i was deleted. During the processing of p_i, if p_j is not yet in the triangulation, we delay the insertion of p_i to a later iteration. This search is done in parallel with one thread handling one missing point.

To avoid concurrent modification of the same triangle during the parallel execution, this phase is also done in multiple iterations; see Algorithm 3. In each iteration, each thread handling an insertion first uses the index of the point to be inserted to mark on the triangles in its container (line 2–5). Next, each thread checks the marks on these triangles and only performs the insertion if its marks are not overwritten by any other threads (line 7). A global synchronization, which is fairly cheap on the GPU, is required to make sure that all the markings are done before any check is performed. The marking is done using the *atomic minimum* operation [21], which is readily available on the GPUs. This guarantees the termination of the algorithm, since in each iteration at least the missing point with the smallest index can be inserted. This also means many iterations may potentially be needed to complete the insertions, since many points may fall into the same triangle. Nevertheless, in practice, after

Algorithm 3 Inserting missing points

```
1: repeat
2:   for each yet-to-be-inserted missing point p_i do in parallel
3:     locate the container of p_i;
4:     mark the triangle(s) containing p_i with i using atomic minimum;
5:   end for
6:   for each yet-to-be-inserted missing point p_i do in parallel
7:     if its container is still marked as i then
8:       mark p_i as active;
9:       mark its container as deleted;
10:    else
11:      record one triangle containing p_i for latter point location;
12:    end if
13:  end for
14:  for each q incident to a deleted triangle do in parallel
15:    Fix q's linked list in the vertex array
16:  end for
17:  for each p marked as active do in parallel
18:    Insert p into its container to create new triangles and update the vertex array;
19:  end for
20:  for each new triangle t do in parallel
21:    Compute t's 3 neighbors and update the links between t and its neighbors;
22:  end for
23: until all missing points have been inserted;
```

the first few iterations, such a triangle is subdivided into many triangles, and the missing points are distributed across them. Thus, the number of iterations is usually small.

For each missing point that can be inserted, we mark its container as deleted (line 8–9). Otherwise, we record its container for later search (line 11). After that, we fix the vertex array of points that are incident to some deleted triangles (line 14–16). Then, we generate new triangles, update their neighbors correspondingly, and update the vertex array of points that are incident to some new triangles (line 17–22), similar to the process in Phase 3. One small implementation note is on the insertion of new triangles into the triangle list in parallel. We have to re-use all the deleted slots in the previous phase to make sure that we do not have to resize the triangle list. This can be done by first collecting the list of deleted slots using parallel stream compaction. Each missing point being inserted needs up to two new slots. We can use parallel prefix sum to allocate the available slots in the triangle list to each thread.

At the end of this phase, we can remove the fake boundary introduced in Phase 2. This is done in the same manner as deleting a bad case in Phase 3, except that each hole created when deleting a boundary point is an open polygon (still star-shaped). After the fake boundary is removed, we compact the triangle list to remove all deleted slots that are not used. This changes the index of the triangles, so we have to update all the references to them.

3.5 Phase 5: Edge flipping

This phase transforms the current triangulation into the DT. We verify the empty circle property of each edge in our triangulation in parallel. For an edge ab of triangle abc, we check if the point d of the adjacent triangle adb is inside the circumcircle of abc. If so, an edge flipping is performed to replace abc and adb with adc and cdb. This process is done in multiple iterations, and the same strategy as in the previous phase is used to avoid concurrent modification of a triangle by multiple threads. We use one thread to process one triangle, and we mark those that do not need any flipping so we do not need to check them again in the subsequent iterations. Note that when a pair of triangles is flipped, the new triangles are not marked. Another optimization for this phase is that a triangle t_i only performs the in-circle test with its neighbor t_j if $i < j$, since each pair of triangles should only be checked at most once in each iteration. However, the implementation needs to make sure that as long as one of the triangles in a pair is not marked, the pair should be checked in the next round.

To perform the in-circle test, we use the exact predicate introduced by Shewchuk [27] which is adaptive and robust. Besides the exact predicate, Shewchuk also provides a fast check that is only approximate but fast. We use the fast check on the GPU, while marking the edges that have numerically inaccurate in-circle results (those almost co-circular cases). After all possible flips are done, we perform the checking and flipping of those few marked edges on the CPU using the exact predicate. This is faster than using the exact predicate on the GPU since we transfer the triangulation back to the CPU as output anyway. In case the output DT need not be transfered back, the exact predicate can be done directly on the GPU.

One difficulty during this phase is the updating of the links between the triangles after each flipping iteration. Two adjacent triangles can participate in two different flips, thus directly updating the adjacent triangles after flipping can cause conflicting memory access. Instead, the update is performed in two steps, each is done in parallel, with a global synchronization in between. Each triangle has a temporary storage for updating its links. In the first step, each pair of triangles that is just flipped updates the temporary storage of its neighbors. In the second step, each pair of triangles mentioned above inspects its temporary storage and updates its own links. Note that if a

neighbor of this pair is not flipped in this iteration, that neighbor is not processed by any thread. Thus, the thread processing this pair needs to update that neighbor's links directly.

4 Computing the CDT on the GPU

To introduce constraints into the DT computation, we use the approach of constructing a triangulation $\mathcal{T}$ of the point set first before incorporating the constraints. This is because considering constraints earlier in the digital VD computation makes the dualization much more difficult, and the correctness of the resulting triangulation might not be guaranteed.

A naïve approach is to have one thread handling one constraint, deleting triangles that it pierces through and re-triangulating the created region. This, however, is not ideal because each constraint can intersect a different number of triangles in $\mathcal{T}$, resulting in unbalanced workloads. Furthermore, two different constraints may intersect some common triangles, and the threads handling them cannot proceed without some sort of locking, a costly operation on the GPU.

To achieve good parallelism, we employ the flipping approach to insert constraints. Multiple pairs of triangles intersected by the same constraint can possibly be flipped in parallel. Also, when multiple constraints intersect some common triangles, we can still possibly flip some of them. To regularize work among different threads, this flipping is done before Phase 5 of the DT algorithm in Section 3, so that we can focus on inserting the constraints first before worrying about the empty circle property. Our algorithm can be summarized as follows:

Step 1. Compute a triangulation $\mathcal{T}$ for all points (Phases 1 to 4);
Step 2. Insert constraints into $\mathcal{T}$ in parallel;
Step 3. Verify the empty circle property for each edge (that is not a constraint), and perform edge flipping if necessary.

Step 3 is similar to Phase 5 of the DT algorithm, with some slight modifications to not flipping constraints. Our proposed Step 2, with an *outer loop* and an *inner loop*, is given in Algorithm 4. The idea is to identify intersections between constraints and triangles, i.e. constraint-triangle intersections, in the outer loop, and to use edge flipping to remove them in the inner loop, all in parallel using multiple passes.

Algorithm 4 Inserting constraints into the triangulation

```
1: repeat    /* outer loop */
2:   for each constraint c_i do in parallel
3:     mark triangles intersecting c_i with i using
       atomic minimum;
4:   end for
5:   repeat    /* inner loop : see Algorithm 5 */
6:     do edge flipping to remove intersections to
       constraints;
7:   until no edge is flippable;
8: until all constraints are inserted;
```

4.1 Outer loop: Find constraint-triangle intersections

For each triangle in the triangulation, we find the index of one constraint intersecting it, if any. Let $c_i = ab$ be the i^{th} constraint in the input, we go through the triangle fan of a to identify the triangle A intersected by c_i. If c_i is an edge of A, the constraint is already there in the triangulation and no further processing is needed. Otherwise, from A we start walking along the constraint towards b, visiting all triangles intersected by c_i. For each triangle found, we mark it with the index i using the atomic minimum operation. Letting the minimum index remain as the marker is necessary for our proof of correctness. Since we do not modify anything in the triangulation in this step, no locking is needed. The work done in this outer loop achieves coarse-grained parallelism on the GPU, with one thread processing one constraint.

4.2 Inner loop: Remove intersections

The inner loop of Algorithm 4 performs edge flipping to reduce the number of constraint-triangle intersections. Here, the parallelism is fine-grained with each thread processing a triangle. Consider a pair of adjacent triangles that are both marked by the same constraint. There are four kinds of configuration for it; see Figure 4. A pair is classified as a *zero intersection*, *single intersection* or *double intersection* configuration if flipping it results in a new pair having zero, one or two intersections with the constraint, respectively. If the flipping is not allowed as its underlying space is a concave quadrilateral, the pair is classified as a *concave* configuration.

It might seem good to avoid flipping a double intersection configuration since flipping does not "improve" the situation and thus possibly leading into an infinite loop. However, we note that restricting flipping to only zero and single intersection is not

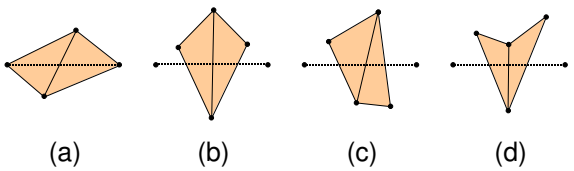


Fig. 4. Configurations of a triangle pair intersecting a constraint (drawn in dashed line). (a) Zero intersection. (b) Single intersection. (c) Double intersection. (d) Concave.

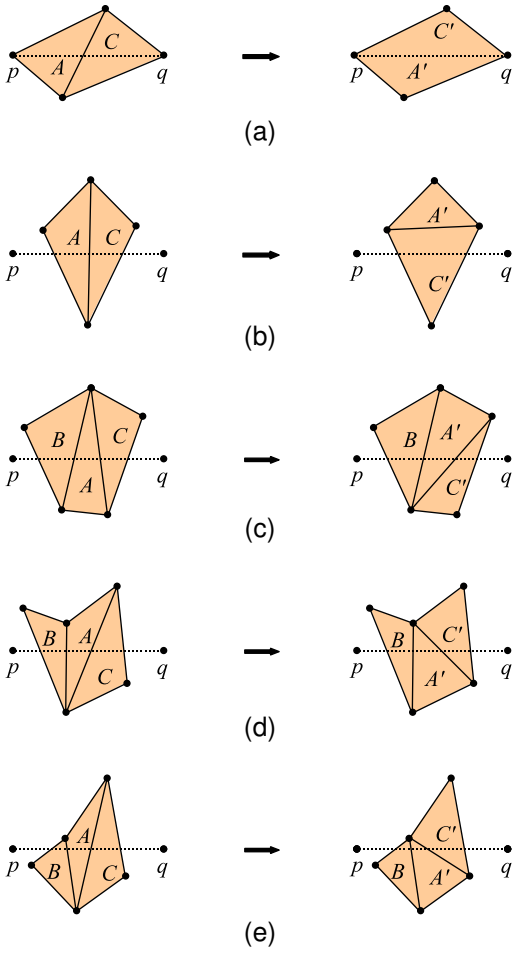


Fig. 5. Flipping consideration of a triangle pair involving A. The constraint pq intersects the triangles from left to right. (a) Case 1a. (b) Case 1b. (c) Case 2. (d) Case 3a. (e) Case 3b.

sufficient to get rid of all the constraint-triangle intersections. To overcome this dilemma, we propose a one-step look-ahead strategy. Consider a triangle A in the chain of triangles intersected by a constraint, from one end point to the other, and let B and C be the previous and the next triangle in that chain. The triangle pair (A, C) is *flippable* in one of the following cases: (Figure 5)

Case 1. (A, C) is a zero or single intersection.

Case 2. (A, C) and (B, A) are both double intersections, and flipping (A, C) would result in B with its new next triangle forming a single intersection.

Case 3. (A, C) is a double intersection and (B, A) is concave, and flipping (A, C) would result in B with its new next triangle no longer being concave.

Note that Case 2 is equivalent to $(B \cup A \cup C)$ being a convex polygon. We perform the flipping in multiple iterations; see Algorithm 5. In each iteration, we first identify the triangle pairs and their configurations

Algorithm 5 Processing of constraint-triangle intersections

```
1:  repeat
2:    for each triangle A intersecting a constraint do in parallel
3:      if C is also marked by the same constraint as A then
4:        determine the case of (A, C);
5:      end if
6:    end for
7:    for each triangle A intersecting a constraint do in parallel
8:      if (A, C) is flippable then
9:        mark A, C (and B for Case 2 and Case 3) using atomic minimum;
10:     end if
11:   end for
12:   for each triangle A intersecting a constraint do in parallel
13:     if A, C (and B for Case 2 and Case 3) retain the same mark then
14:       flip (A, C) and update the links between the new triangles and their neighbors;
15:     end if
16:   end for
17: until no edge is flippable;
```

(line 2–6). Then for any flippable pair (A, C) as described above, we mark A, C, and possibly the previous triangle of A (which is B in our discussion), with the index of A using the atomic minimum operation (line 7–11). Lastly, we flip (A, C) only if the marks on A and C (and also B for Case 2 and Case 3) remain (line 12–16). This prevents the possible conflicts when updating the triangulation, and allows the one-step look-ahead to be achieved. We also introduce extra weights into the labels used in the marking. Case 1 is given the highest weight, followed by Case 2 and then Case 3. For each step to be done in parallel, we assign one thread to process one triangle. As an optimization, after each iteration, we maintain a compact list of *active* triangles, i.e. those that are still intersected by the constraints. As such, in later iterations, we do not have too many idle threads handling triangles that no longer active.

In practice, the **repeat-until** loop in Algorithm 5 is only executed a few times per outer loop iteration instead of repeating until no edge is flippable. The reason is as the algorithm progresses, there is a drastic reduction in the number of flippable cases, and thus the parallelism reduces. By switching to the outer loop after a few (say 5 to 10) iterations of the inner loop, the algorithm can discover more flippable cases to improve the parallelism and as a result improving the performance, without compromising on the correctness proven in the next section.

4.3 Proof of correctness

We show that Algorithm 4 terminates with all constraints inserted into the triangulation. Consider one iteration of the outer loop, and let $c_i = ab$ be the constraint with the smallest index i that still intersects some triangles in our triangulation. By using the atomic minimum operation, we ensure that all triangles intersecting c_i are marked with i. It thus suffices to prove the following:

Claim 1. The inner loop always successfully inserts one constraint into the triangulation.

Proof. Consider the chain of triangles intersecting c_i from a to b. Among these triangles, if there are one or more triangle pairs that are single or zero intersection, then the claim is true as the marking favors each of these cases and flipping is indeed carried out, reducing the number of triangles intersecting c_i. Otherwise, consider the chain of triangles having only double intersection or concave configurations. We argue that there exists a triangle pair (A, C) among them that is flippable, and each flip is a step closer to removing the intersections of the triangles with c_i.

If we would remove all triangles intersecting c_i, a polygonal hole is created with points $p_1, p_2, \ldots$ as its upper part and $q_1, q_2, \ldots$ as its lower part, excluding a and b; see Figure 6(a).

Any polygon has an ear. Let $q_{k-1}q_kq_{k+1}$ be the ear such that the triangle $C = q_kq_{k+1}p_j$ incident to q_kq_{k+1} and intersected by ab is the earliest one in the chain. We exclude a and b themselves to be q_k. Let A be the previous triangle of C and B be the previous of A. We have $A = q_kp_jp_{j-1}$ since if it were $q_kp_jq_{k-1}$, (A, C) would have been a single intersection pair. The triangle pair (A, C) is a double intersection, since the two angles $p_{j-1}p_jq_{k+1}$ and $p_{j-1}q_kq_{k+1}$ are both less than π.

We claim that (A, C) is flippable by considering 3 cases of B as follows. If $B = q_kp_{j-1}p_{j-2}$, (B, A) is a concave pair; see Figure 6(b). $p_{j-2}p_{j-1}q_{k+1}q_k$ is convex by the choice of q_k, thus triangles B, A, C fulfill Case 3a, and (A, C) is flippable. If $B = q_kp_{j-1}q_{k-1}$ and (B, A) is a double intersection (see Figure 6(c)), the union of triangles B, A, C is a convex polygon as needed in Case 2, so (A, C) is flippable. If $B = q_kp_{j-1}q_{k-1}$ and (B, A) is a concave configuration (see Figure 6(d)), the union of triangles B, A, C fulfill Case 3b, so (A, C) is also flippable. As long as there is one flippable triangle pair, the marking in the inner loop in Algorithm 4 will successfully mark one for flipping, and flipping is indeed performed in each pass.

We next show that our inner loop does not go on forever. Let us assign to each pair of triangles a value of 0, 1 or 2. A pair of triangles that is zero or single intersection is assigned value 0; a double intersection, value 1; and a concave, a value 2. As a result, we have a base 3 number, N, to record the cases of the chain of triangles intersecting c_i. A flip due to Case 1 deletes a digit in N; Case 2 turns 11 into 01; and Case 3 turns 21 into 11 (Case 3a) or 01 (Case 3b). In other words, each flip decreases the value of N. Since N is finite, our algorithm clearly terminates, and a constraint is inserted as claimed. □

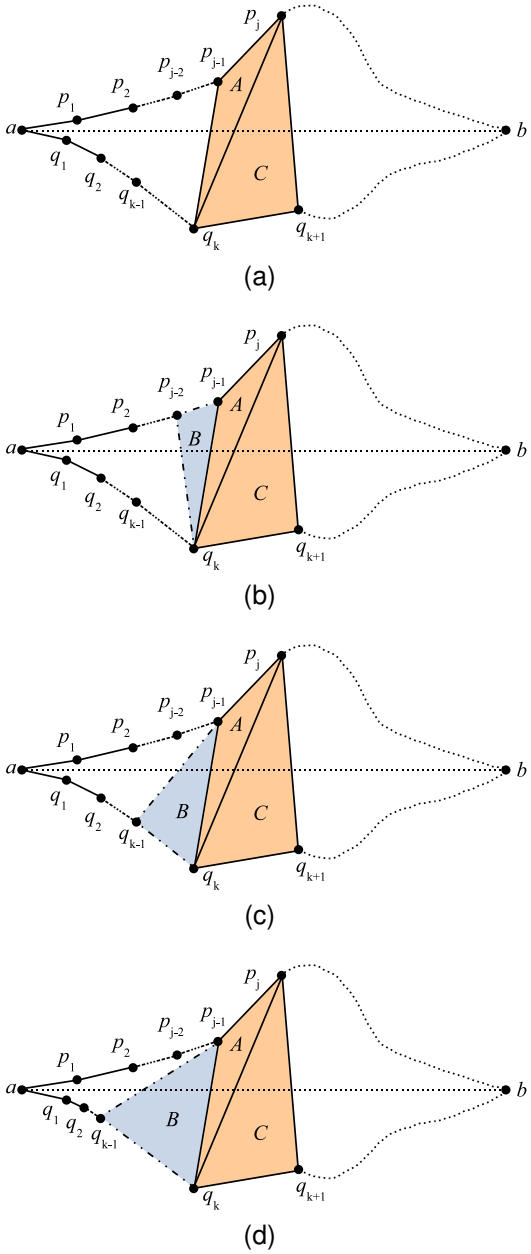


Fig. 6. (a) When the triangle pairs intersecting the constraint $c_i = ab$ are only either double intersection or concave, there exists a flippable pair (A, C). (b) B, A and C fulfill Case 3a. (c) B, A and C fulfill Case 2. (d) B, A and C fulfill Case 3b.

4.4 Complexity analysis

Claim 1 concludes that our proposed algorithm computes the CDT correctly. In this subsection, we show that no flip is wasteful. We first analyze the number

of flips needed to insert one constraint, followed by a bound on the total number of flips needed to insert all the constraints.

Claim 2. The total number of flips performed by the inner loop in order to insert one constraint is $O(k^2)$, where k is the number of triangles intersecting the constraint.

Proof. Flipping due to Case 1 cannot be done more than k times since each flipping removes an intersection. Flipping due to Case 2 immediately gives rise to a flipping of Case 1 (with highest priority), and thus also cannot be done more than k times.

The initial number of concave pairs is bounded by $O(k)$. A flip due to Case 1 (or Case 2) introduces at most two (or one) concave pairs, thus at most $O(k)$ concave pairs can be introduced by these two flipping cases. Flipping due to Case 3 either eliminates a concave pair or pushes it towards one end of the constraint, thus it can be performed no more than $O(k^2)$ times. As a result, the total number of flips is $O(k^2)$. □

Figure 7(a) shows that the worst case of Claim 2 can happen. In this example, among all the triangles intersected by the constraint, there is a particular chain of triangles with k successive concave pairs followed by k double intersection pairs, such that none of the double intersection pairs fulfill Case 2. The only flippable case in this situation is due to Case 3a with two triangles sharing the edge ef. Flipping this pair of triangles and moving on with another $k-1$ flippings due to Case 3a, the algorithm produces Figure 7(b). The concave pair on the left of the k double intersections has been "removed" and "introduced" in the right at the edge st. In other words, the concave pair is *pushed* towards one end of the constraint. As such, for each concave pair, $O(k)$ flips are needed to remove it, and since there are k concave pairs being removed in parallel, we need $O(k)$ iterations of the inner loop and $O(k^2)$ flips are performed.

One technicality remains in the above construction. We add an extra $O(k)$ concave pairs on the left and $O(k)$ double intersection pairs on the right of the chain of triangles shown in Figure 7(a). This is to make sure that the triangle pairs incident to a and b can be flipped $O(k)$ times in the $O(k)$ inner loop iterations without affecting the chain of concave and double intersection pairs shown in the figure. The total number of triangles intersecting the constraint is $O(k)$, so our $O(k^2)$ bound on the number of flips needed is tight.

Following the previous claim, in the worst case the total number of flips performed to insert all the constraints may reach $O(n^3)$. However, this bound is not tight. In the next claim, we show that the total work for inserting all constraints is $\Theta(n^2)$.

Claim 3. The total number of flips performed by Algorithm 4 is $\Theta(n^2)$, where n is the number of input points.

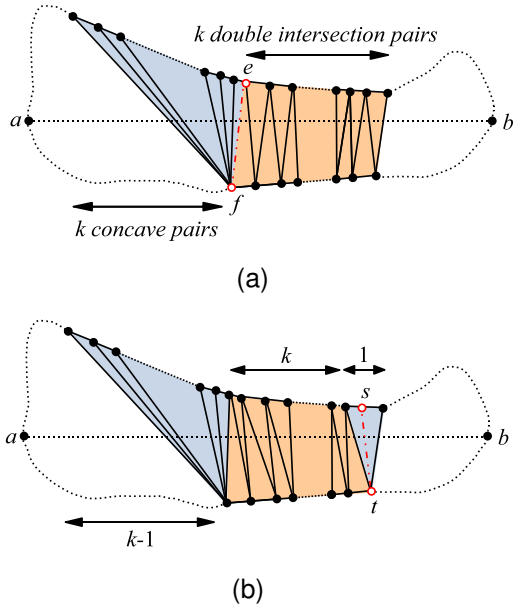


Fig. 7. A bad case for inserting one constraint.

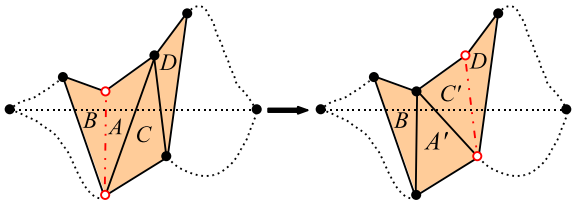


Fig. 8. Push the concave pair towards the right end of the constraint using a flipping due to Case 3a.

Proof. For any constraint c_i, if all triangles intersecting it are deleted, a polygonal region is left with s_i points in its upper part and t_i points in the lower part, excluding the endpoints of c_i. The number of triangles intersecting c_i is $k = s_i + t_i$.

According to Claim 2, flipping due to all cases except Case 3a can only be done $O(k)$ times. Flipping due to Case 3a either eliminates a concave pair or pushes it towards the right end of the constraint. When a concave pair is pushed, it moves one step to the right end in both the upper boundary and the lower boundary of the polygonal region (Figure 8), thus each concave pair can only be pushed $\min(s_i, t_i)$ times. As such, the total number of flips to insert c_i is $O(k + k\ \min(s_i, t_i)) \subset O(s_i t_i)$. The total number of flips performed for m constraints is thus $O(\sum_{i=1}^{m} s_i t_i)$.

Each time a constraint c_i is inserted, any edge $p_i q_j$ can never appear later in the triangulation. Not only that, such an edge also cannot be inside the polygonal region of any other constraint, otherwise that region would have intersected c_i. The number of edges that can possibly be formed by the n input points is $O(n^2)$, so $O(\sum_{i=1}^{m} s_i t_i) \in O(n^2)$. This, together with the worst case example shown above, concludes our proof of Claim 3. □

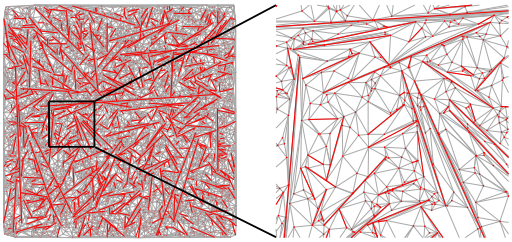


Fig. 9. A synthetic dataset (left) and its constrained Delaunay triangulation (right).

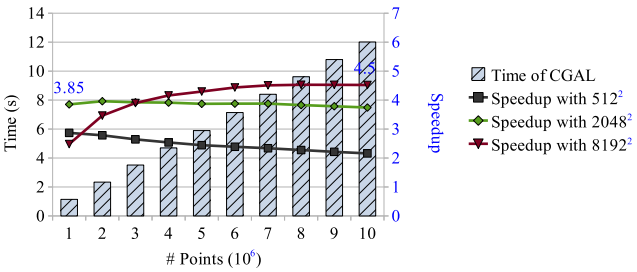


Fig. 10. The running time of CGAL and the speedup of gCD over CGAL on DT computation time.

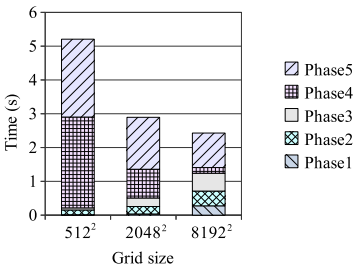


Fig. 11. The running time of different phases of the DT computation with 10M points.

In our experiment in Section 5, we observe that the total number of flips performed to insert all constraints is actually proportional to the number of constraint-triangle intersections.

5 Experiment Results

Our algorithm is implemented using the CUDA programming model by NVIDIA. All the experiments are conducted on a PC with an Intel i7 2600K 3.4GHz CPU, 16GB of DDR3 RAM and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory. Visual Studio 2008 and CUDA 4.0 Toolkit are used to compile all the programs, with all optimizations enabled. To achieve exact and robust result during our computation, we only use orientation and in-circle predicates from the exact predicates of Shewchuk [28].

The input to the program is a PSLG containing possibly no edges. All numbers and computations are done in double precision. To assess the efficiency of our gCD program, we compare its running time, on both synthetic and real-world data, with that of the most popular computational geometry software available, *Triangle* and CGAL version 3.9. According to our tests, CGAL runs faster than *Triangle* for the DT computation. However, when constraints are introduced, *Triangle* runs much faster than CGAL. Here, we only show the result of the faster one between the two.

5.1 Synthetic Dataset

To generate synthetic data, we first randomly generate constraints of different lengths such that none of them intersect each other. Then, we randomly generate points that do not lie on any constraint. Figure 9 shows one small synthetic data generated.

5.1.1 Comparison on DT results

This is the case where the input has no constraints. For different number of input points, our approach achieves 4 to 4.5 times speedup over CGAL; see Figure 10. The results on grid sizes 1024^2 and 4096^2 are omitted in all figures for clarity. As a side note, we also verify that our work is superior to the prior work reported by Rong et al. [25] for the DT computation.

Different grid sizes used for the digital VD computation significantly affect the running time of our program. This can be analyzed more thoroughly from Figure 11 which shows the running time of different phases for computing DT with 10M points. A larger grid has some penalty on the digital VD computation time in Phase 1, and leads to more points (that can be mapped onto the grid) to be shifted in Phase 3. However, it gives a better approximation and less missing points, thus Phase 4 and Phase 5 run faster, contributing to a faster running time overall. All in all, as a general guideline, given a larger set of input points, a larger grid is preferable.

The same conclusion is also true when the input points are from a Gaussian distribution. For small grid sizes, the speedup achieved is slightly lower due to having more points concentrated in the center of the grid and became missing points. The speedup increases quickly when we move to a larger grid. However, this is no longer true on extreme cases such as when points are co-circular, since our uniform digital VD is not a good approximation of the continuous one in this case.

5.1.2 Comparison on CDT results

When constraints are introduced, we observe a substantial speedup, of up to an order of magnitude, compared to both *Triangle* and CGAL (with CGAL being much slower than *Triangle*). *Triangle* inserts constraints one by one on the DT of the point set while also using an edge flipping method. We compare the time for constraints insertion by subtracting the time for

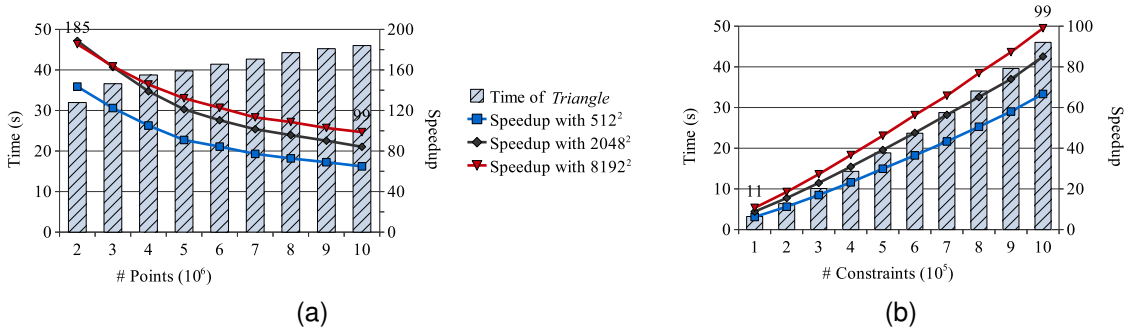


Fig. 12. The running time of *Triangle* and the speedup of gCD over *Triangle* when computing the CDT, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

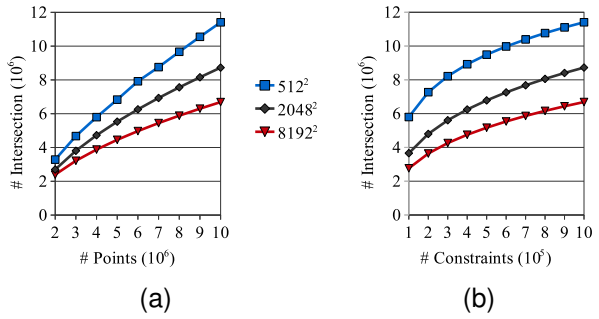


Fig. 13. The total number of constraint-triangle intersections when using different grid sizes, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

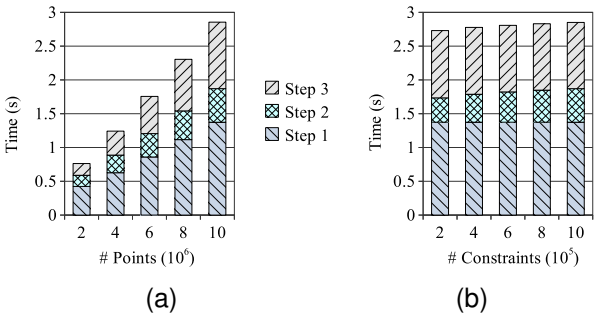


Fig. 14. The running time of different steps of the CDT computation, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

the DT computation from the total time for the CDT computation on the same point set.

Running time: Figure 12(a) and 12(b) show the performance comparison of *Triangle* with gCD on different number of points and constraints, with different grid sizes. Clearly, the more constraints there are, the higher is the speedup we achieve. This is because only a small part of our constraints insertion algorithm is done with coarse-grained parallelism, while the majority of the processing has fine-grained parallelism. As a result, our algorithm scales well with the amount of work available. Note that we achieve better performance for constraints insertion when using bigger grid sizes because the number of constraint-triangle intersections decreases (see Figure 13(a) and 13(b)). This is possibly due to the fact that the triangulation produced by Step 1 is closer to the DT when the grid size gets bigger.

Running time of different phases: Figure 14(a) and 14(b) shows the running time of different steps of gCD using 8192^2 grid size. Similar behavior is also observed for other grid sizes. The time our program spent on inserting constraints occupies less than 20% of the total time. On the same datasets, *Triangle* spends most of its time for that task. For example, given 10M points and 1M constraints, *Triangle* spends 62 seconds for constructing the CDT, of which 46 seconds

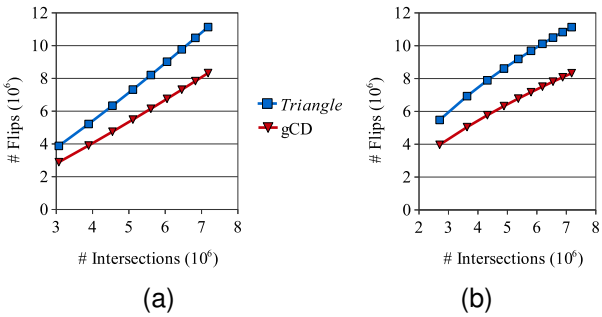


Fig. 15. The number of flips required given different number of constraint-triangle intersections when inserting constraints, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

are spent on constraints insertion. In contrast, gCD spends 3.2 seconds for constructing the CDT, of which only 0.47 seconds are for inserting constraints. As such, when comparing the total running time of our program with that of *Triangle*, we achieve significant speedup, ranges from 10 to 49 times.

Number of flips: We compare with *Triangle* on the number of flips needed to insert constraints. To get a fair comparison, we follow *Triangle*'s approach by modifying our implementation to insert the con-

TABLE 1
Running time on the contour datasets.

Set	Data Size	Constraints Insertion (s)		Speedup
		Triangle	gCD	
a	1.2M	0.665	0.046	14×
b	3.2M	1.982	0.071	28×
c	4.5M	2.526	0.097	26×
d	5.7M	3.181	0.133	24×
e	8.5M	4.755	0.245	19×
f	9.5M	6.036	0.244	24×

straints after we have obtained the DT. The flips needed by gCD to insert constraints include both those to make the constraints appear in the triangulation and those to make the triangulation a CDT.

Figure 15(a) and 15(b) shows the comparison with different number of points and constraints. The number of flips is plotted against the number of constraint-triangle intersections to highlight that the relationship between the two is linear in practice. This is much lower than the worst case complexity analyzed in Section 4.4. Note that we perform slightly less edge flippings than *Triangle*, possibly due to our algorithm giving extra weights to Case 1 and Case 2, which are the cases that lead to an immediate removal of some constraint-triangle intersections.

5.2 Real-world datasets

To compare gCD and *Triangle* on real-world datasets, we use the contour maps freely available at https://www.ga.gov.au/. Figure 1(a) shows an example of the contour maps we use in our experiment, together with its CDT. For such a contour map, the number of constraints is approximately the same as the number of points. Here we use the number of points to denote the data size of a contour map. Table 1 shows the running time comparison for several different datasets.

Generally, gCD runs faster than *Triangle*. In these real-world datasets, most constraints are very short and do not intersect many triangles (if at all). Figure 16 shows the distribution of the number of intersections per constraint, collected by our program for the Dataset *f* (with about 9.5M points and 9.5M constraints) and for a synthetic dataset (with 10M points and 1M constraints). The maximum number of intersections per constraint is 51 for the contour dataset as compared to 7073 for the synthetic dataset.

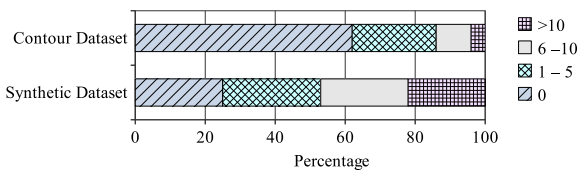


Fig. 16. The distribution of the number of intersections per constraint.

In both cases, the total number of constraint-triangle intersections is around 6M. *Triangle* inserts constraints much slower when the constraints are long, i.e. one constraint intersects many triangles. It takes 46 seconds for the synthetic dataset, while only 6 seconds for the contour dataset where constraints are mostly short. On the other hand, due to our fine-grained approach, our program can easily process the synthetic dataset consisting of both long and short constraints. The parallelism achieved is similar to when processing the contour dataset having all constraints being very short. Our running time for synthetic and contour dataset are 0.47 and 0.25 seconds respectively, showing a significant speedup over the sequential approach.

5.3 Image vectorization

As mentioned earlier, the CDT can be used in many applications. Prasad and Skourikhine [22] present a framework for transforming a raster image into a vector image comprised of polygons that can be subsequently used in image analysis. Their algorithm consists of the following steps. First, edges are recognized using some standard edge detection algorithm. Second, contiguous edge pixel chains are extracted and an edge map consisting of many straight line segments is constructed. Third, the CDT of the edge map (which is actually a PSLG) is computed. Finally, adjacent *trixels* (triangles) are merged based on some certain grouping filters, and the connected components of the trixel grouping graph yield polygons that represent the image. Figure 1(b) shows a raster image and the result after computing the CDT of its edge map. In practice, depending on the resolution of the image, the edge map might consist of hundreds of thousands of line segments (constraints) of different lengths, thus using our gCD help speed up the computation. The constraints here are similar in nature to those in the contour datasets, so the performance of gCD is also similar.

6 Concluding Remarks

This paper presents a new, efficient and robust parallel approach to construct the 2D constrained Delaunay triangulation on the GPU. Our approach is scalable and is capable of maximizing the parallelism to utilize the computing power of the GPU. That has been shown in our experiment with both synthetic and real-world data. Our implementation achieves an order of magnitude better performance than the best CPU libraries available. A limitation of our approach is that we construct the DT using the digital VD computed on a uniform grid. As a result, our algorithm is fast when the input points are uniformly distributed, while performing less efficiently when the input point

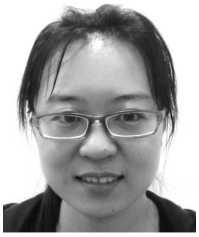
distribution is highly skewed. Nevertheless, we believe our approach is useful for many practical applications.

ACKNOWLEDGMENTS

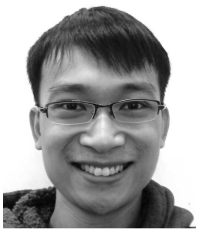
This research is supported by the National University of Singapore under grant R-252-000-337-112.

REFERENCES

[1] F. Aurenhammer, "Voronoi diagrams – a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.
[2] J. Bernal, "Inserting line segments into triangulations and tetrahedralizations," National Institute of Standards and Technology, Tech. Rep. 5596, 1995.
[3] J.-D. Boissonnat, "Shape reconstruction from planar cross sections," *Computer Vision, Graphics, and Image Processing*, vol. 44, no. 1, pp. 1–29, 1988.
[4] T.-T. Cao, H. Edelsbrunner, and T.-S. Tan, "Proof of correctness of the digital Delaunay triangulation algorithm," 2011, http://www.comp.nus.edu.sg/~tants/delaunay2DDownload_files/notes-30-april-2011.pdf.
[5] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the GPU," in *I3D '10: Proc. ACM Symp. Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2010, pp. 83–90.
[6] CGAL, "CGAL, Computational Geometry Algorithms Library," http://www.cgal.org, 2011.
[7] L. P. Chew, "Constrained Delaunay triangulations," *Algorithmica*, vol. 4, pp. 97–108, 1989.
[8] V. Domiter, "Constrained Delaunay triangulation using plane subdivision," in *Proceedings of the 8th central European seminar on computer graphics*, 2004, pp. 105–110.
[9] R. Dwyer, "A faster divide-and-conquer algorithm for constructing delaunay triangulations," *Algorithmica*, vol. 2, pp. 137–151, 1987.
[10] I. Fischer and C. Gotsman, "Fast approximation of high-order voronoi diagrams and distance transforms on the GPU," *J. Graphics Tools*, vol. 11, no. 4, pp. 39–60, 2006.
[11] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, pp. 153–174, 1987.
[12] ——, *Handbook of discrete and computational geometry*. Boca Raton, FL, USA: CRC Press, Inc., 1997, ch. Voronoi diagrams and Delaunay triangulations, pp. 377–388.
[13] C. M. Gold, "A review of potential applications of voronoi methods in geomatics," in *Proceedings of the Canadian Conference on GIS*, 1994, pp. 1647–1656.
[14] R. L. Graham, "An efficient algorith for determining the convex hull of a finite planar set," *Information Processing Letters*, vol. 1, no. 4, pp. 132 – 133, 1972.
[15] L. Guibas, D. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, pp. 381–413, 1992.
[16] P. T. Highnam, "The ears of a polygon," in *Information Processing Letters*, 1982, pp. 196–198.
[17] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized Voronoi diagrams using graphics hardware," in *Proc. ACM SIGGRAPH '99*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286.
[18] K. H. Huebner, D. L. Dewhirst, D. E. Smith, and T. G. Byrom, *The Finite Element Method for Engineers*. New York, NY, USA: Wiley, 2001.
[19] M. Kallmann, "Shortest paths with arbitrary clearance from navigation meshes," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 159–168.
[20] D. Lee and A. Lin, "Generalized Delaunay triangulation for planar graphs," *Discrete and Computational Geometry*, vol. 1, pp. 201–217, 1986.
[21] NVIDIA, "NVIDIA CUDA C Programming Guide," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.
[22] L. Prasad and A. N. Skourikhine, "Vectorized image segmentation via trixel agglomeration," *Pattern Recognition*, vol. 39, pp. 501–514, April 2006.
[23] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1985.
[24] 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*, ser. I3D '12. New York, NY, USA: ACM, 2012, pp. 39–46.
[25] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus, "Computing two-dimensional Delaunay triangulation using graphics hardware," in *I3D '08: Proc. Symp. Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2008, pp. 89–97.
[26] M. I. Shamos and D. Hoey, "Closest-point problems," *FOCS '75*, pp. 151–162, 1975.
[27] J. Shewchuk, "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator," in *Applied Computational Geometry Towards Geometric Engineering*, ser. Lecture Notes in Computer Science, M. Lin and D. Manocha, Eds. Springer Berlin / Heidelberg, 1996, vol. 1148, pp. 203–222.
[28] J. R. Shewchuk, "Robust adaptive floating-point geometric predicates," ser. SoCG '96. New York, NY, USA: ACM, 1996, pp. 141–150.
[29] P. Su and R. L. Scot Drysdale, "A comparison of sequential Delaunay triangulation algorithms," *Computational Geometry: Theory and Applications*, vol. 7, pp. 361–385, April 1997.
[30] L. A. Treinish, "Visualization of scattered meteorological data," *IEEE Computer Graphics and Applications*, vol. 15, pp. 20–26, 1995.



Meng Qi is a PhD student at the School of Computing, National University of Singapore. She is interested in using the GPU to solve the two-dimensional mesh generation problem.



Thanh-Tung Cao is a PhD student at the School of Computing, National University of Singapore. His interest is in solving computational geometry problems using parallel algorithms, especially on the GPU.

Tiow-Seng Tan is an Associate Professor at the School of Computing, National University of Singapore. He 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.

APPENDIX A
OUR FLOODING ALGORITHM

Phase 1 of our proposed algorithm is to compute a digital VD of a collection of points (or *seeds*) in a grid. Technically, we want and will prove in the remaining part of this appendix that the output of this phase is the same as that of the Standard flooding mentioned in [1]. Then, we can apply their result to conclude that Phase 2 of our algorithm indeed produces a triangulation.

In our gCD algorithm, we use the efficient Parallel Banding Algorithm (PBA) [2] on the GPU to start the coloring, which results in an Euclidean coloring. Each of the Voronoi region obtained has a connected component called *bulk* which is path-connected to its seed, and *debris* (if any) which are disconnected from the seed (see Figure A.1). Cao et al. [1] show that bulks are subsets of the result of the Standard flooding.

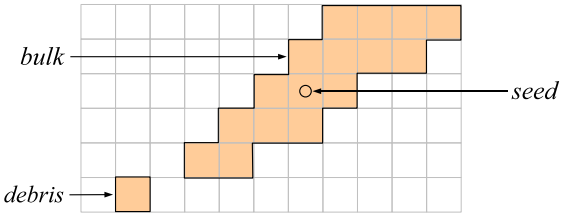


Fig. A.1. Illustration for one seed and its debris and bulk.

Therefore, our challenge is to identify and recolor the debris to be the same as in the Standard flooding result. Since there are very few debris in general, the recoloring is done on the CPU and the result is copied to the GPU to complete Phase 1. Algorithm A.1 shows our proposed approach. We use Q to refer to a priority queue, $N(A)$ the set of grid points neighboring grid point A, and p_i a seed with color i. The operation $\min(Q)$ on Q removes and returns the pair with minimum distance between the grid point and the seed of its color; $\|A-p_i\| \prec \|B-p_j\|$ means $\|A-p_i\| < \|B-p_j\|$, or $\|A-p_i\| = \|B-p_j\|$ with consistent tie breaker (using, for example, the coordinates of the points).

We now show by contradiction that Algorithm A.1 indeed produces the same output as that of the Standard flooding. We just need to argue for those grid points that are identified as debris since bulks are shown to be the same already [1]. Consider the very first instance when our algorithm colors a debris A with color r (inside the while-loop) whereas the Standard flooding produces grid point A with color $s \neq r$. There are two situations to consider but both lead to a contradiction and thus no such debris exists:

Case 1: $\|A-p_r\| \prec \|A-p_s\|$.
From Algorithm A.1, there exists a neighbor B of A colored by r earlier, and $\|B-p_r\| \prec \|A-p_r\|$.

Algorithm A.1 Our Flooding Algorithm

```
1: Compute coloring with PBA;
2: Identify all debris as uncolor;
3: Q ← ∅;
4: for each debris A do
5:    Q ← Q ∪ {(A, i) | ∃B ∈ N(A) having been colored i and ‖B − p_i‖ ≺ ‖A − p_i‖};
6: end for
7: while Q ≠ ∅ do
8:    (A, i) ← min(Q);
9:    if A is not colored then
10:      Color A with color i;
11:      Q ← Q ∪ {(C, i) | C ∈ N(A) and ‖A − p_i‖ ≺ ‖C − p_i‖};
12:   end if
13: end while
```

It follows that $\|B-p_r\| \prec \|A-p_r\| \prec \|A-p_s\|$. According to our choice of A, the grid point B is colored by r in the Standard flooding. By the Ordered Coloring Lemma [1], (A, r) must have been considered in the Standard flooding before (A, s) and A should thus have been colored by r, a contradiction.

Case 2: $\|A-p_s\| \prec \|A-p_r\|$.
In the result of the Standard flooding, there is a monotonic path from p_s to A. A part of this path has been colored the same (with s) in our flooding algorithm when we are about to color A. Let C be the grid point closest to p_s in that path that is not yet colored by our flooding algorithm. With the previous grid point before C having been colored with s, (C, s) must have been added to Q in our flooding algorithm. Since $\|C-p_s\| \leq \|A-p_s\| \prec \|A-p_r\|$, we must have (C, s) inside Q to be extracted before (A, r), a contradiction.

The argument in Case 2 also implies that the algorithm colors all grid points. This concludes our claim of correctness of our flooding algorithm.

APPENDIX B
TRANSFORMING THE POINT SET

To work on points with floating point coordinates, Phase 1 needs to map them to an $m \times m$ grid. We want precise computation so that the triangulation computed with respect to points mapped to the grid remains a triangulation when we do a part of the inverse mapping to the original coordinates of the points. We discuss below that precise computation can be achieved by representing the *scale* and *translation* used in the mapping with a certain number of bits.

We just consider the 1D coordinate in the x-axis; the discussion can be generalized to 2D with *scale* be the larger one calculated from each dimension, while *translation* is simply a vector of two components. Let

the points be such that their minimum and maximum x-coordinates are $x_{\min}$ and $x_{\max}$, respectively. Let x be the original coordinate of a point. The coordinate of the point mapped to the grid is thus $\bar{x} = \lfloor (x - \textit{translation})/\textit{scale} \rfloor$ where $\textit{translation} = x_{\min}$ and $\textit{scale} = (x_{\max} - x_{\min})/m$. The computation of a triangulation in Phase 1 and Phase 2 is performed using these integer coordinates.

Then, Phase 3 shifts all points in the grid back to their positions given in the input. To maintain as many shifting of good cases as possible, we first perform the inverse scaling and shifting for the whole bounding box with all the points. Specifically, we have $x' = (\bar{x} \times \textit{scale} + \textit{translation})$ as the new coordinate of the point before shifting it to the original coordinate x. The later shifting is only to negate the effect of the truncation to integer coordinate. To ensure we still have the same triangulation with x' in place of $\bar{x}$, we must compute the floating point number x' with no rounding error.

Let $(p\text{Max} + 1)$ be the maximum number of bits available for the mantissa in our floating point representation. Note that the explicit mention of "$+1$" here is a provision for possible overflow of $(\bar{x} \times \textit{scale} + \textit{translation})$. As $\bar{x}$ is a non-negative integer with maximum value of $(m-1)$, it needs $pM = (\log m)$ bits to represent. Let the number of bits used to represent the mantissas of the two constants *scale* and *translation* be pS and pT, respectively. We keep $pT = p\text{Max}$, and mention pS in the next paragraph.

We are ready to discuss how to set *scale* and *translation* before doing the actual mapping to the grid. First, the result of the $(\bar{x} \times \textit{scale})$ is accurately represented using no more than pMax bits, as long as we do some necessary rounding up on *scale* such that its mantissa is representable by $pS = (p\text{Max} - pM)$ bits. The rounding up can just increase *scale* by a little bit at its least significant bits and thus we are still able to spread out the mapping of points on the grid. Second, the addition of $(\bar{x} \times \textit{scale})$ with *translation* can result in rounding error as *translation* can be much smaller or much larger than $(\bar{x} \times \textit{scale})$. Let $\textit{range} = (x_{\max} - x_{\min}) = (m \times \textit{scale})$. We consider two cases to guarantee that the computation of x' is accurate:

Case 1: $\textit{translation} \leq \textit{range}$.
Let 2^t be the largest term in the binary representation of *range*. We reduce *translate* by removing all terms in its binary representation that are smaller than $2^{t-(p\text{Max}-1)}$.

Case 2: $\textit{translation} > \textit{range}$.
We round up the *scale* a little bit as follows. Let 2^r be the largest term in the binary representation of *translation*. We round up all terms that are smaller than $2^{r-(p\text{Max}-1)+pM}$ in the binary representation of *scale*. Because $\textit{range} = x_{\max} - x_{\min} \geq 2^{r-(p\text{Max}-1)}$, we always have that *scale*, represented by pS bits, is larger than $2^{r-(p\text{Max}-1)+pM}$ for any meaningful input and is thus non-zero. Also, the rounding up does not increase the value of *scale* more than twice, and thus we are still able to spread out the mapping of points on the grid.

REFERENCES

[1] T.-T. Cao, H. Edelsbrunner, and T.-S. Tan, "Proof of correctness of the digital Delaunay triangulation algorithm," 2011, http://www.comp.nus.edu.sg/~tants/delaunay2DDownload_files/notes-30-april-2011.pdf.

[2] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the GPU," in *I3D '10: Proc. ACM Symp. Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2010, pp. 83–90.