Computing 2D Constrained Delaunay Triangulation Using the GPU[*]

Meng Qi Thanh-Tung Cao Tiow-Seng Tan

School of Computing
National University of Singapore

Abstract

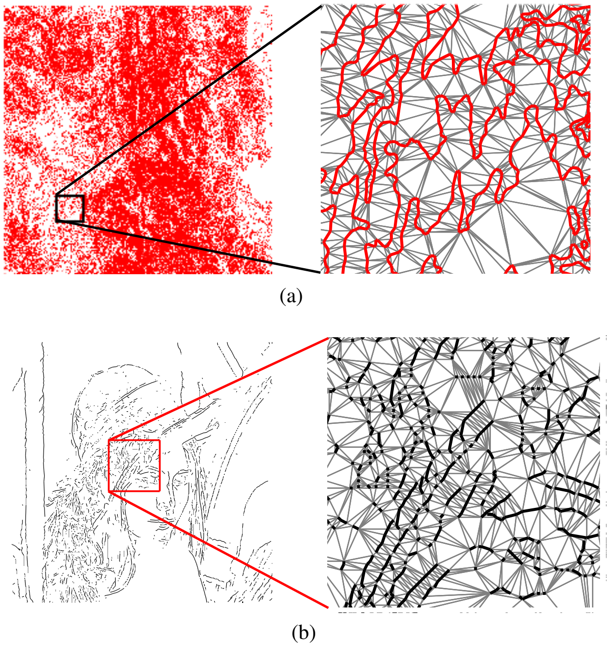
We propose the first GPU solution to compute the 2D constrained Delaunay triangulation (CDT) of a planar straight line graph (PSLG) consisting of points and edges. There are many CPU algorithms developed to solve the CDT problem in computational geometry, yet there has been no known prior approach using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

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

Keywords: GPGPU, Computational Geometry, Voronoi Diagram

1 Introduction

Delaunay triangulation (DT) is one of the most important geometric structures. Due to its nice property of avoiding long, skinny triangles, the DT has many practical applications in different fields. The constrained Delaunay triangulation is a direct extension of the Delaunay triangulation where some edges in the output are enforced before hand [Chew 1989]; 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 constrained Delaunay triangulation (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, boundaries between cities; in surface reconstruction, contours in the slices of the body's skull; in modeling, characteristic curves [Boissonnat 1988; Kallmann et al. 2003]. In short, CDT is a very useful structure in many fields, complementing the DT; see Figure 1.


(a)

(b)

Figure 1: *CDT applications. (a) A contour map. (b) An edge map.*

Recently, the graphics processing unit (GPU) with its enormous parallel computing power has been used for general purpose computation in many disciplines, including computational geometry. Early works include computing the digital Voronoi diagram (VD) [Hoff et al. 1999; Fischer and Gotsman 2006; Cao et al. 2010b], a structure that is closely related to the DT. Recently Rong et al. [2008] present a serious attempt to derive the DT from the digital VD. Their algorithm, however, is hybrid, where parallel computation is only used in the first part, leaving the rest to a sequential CPU algorithm. As for the CDT problem, there is no efficient GPU algorithm as far as we know. In another view, the DT problem, as well as the CDT problem, does not present itself readily to parallel computation. Specifically, it is not clear how to adapt traditional complex parallel techniques, such as divide-and-conquer, while achieving regularized work and localized data access to best utilize the computing power of the GPU.

Our main contribution here is a novel algorithm, termed *GPU-CDT*, to compute the CDT for a given PSLG, fully parallelized on the GPU. Our experiment shows that our implementation using the CUDA programming model is robust and efficient. Comparing to popular software such as *Triangle* [Shewchuk 1996a] and CGAL [CGAL 2011], as well as to the hybrid approach of Rong et al. [2008], GPU-CDT runs up to an order of magnitude faster.

Section 2 introduces some basic 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. Experimental results and applications are described in Section 5. Finally, Section 6 concludes the paper.

[*] Research is supported by the National University of Singapore under R-252-000-337-112. Project website: http://www.comp.nus.edu.sg/~tants/cdt.html. Emails: {qimeng | caothanh | tants}@comp.nus.edu.sg.

ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, March 2012, pp. 39-46.

2 Preliminaries

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the Euclidean space $\mathbb{R}^2$. A *planar straight line graph* (PSLG) $\mathcal{G} = (S, E)$ is a plane graph with vertex set S and 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 points of S are on grid points (which are centers of 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 x is of equal distance from two points p_i and p_j with $i < j$, we color x by p_i. The grid with all grid points colored is called the* digital Voronoi diagram *(VD) of S. This coloring procedure is referred to as the* Euclidean coloring.

Definition 2 (Delaunay Triangulation) *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 points in S are not 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 those points of S visible from both a and b. If $E = \emptyset$, then the CDT of $\mathcal{G}$ is exactly the same as the DT of $\mathcal{G}$.*

There are many sequential algorithms developed for the CPU to compute the DT [Aurenhammer 1991; Fortune 1997; Su and Scot Drysdale 1997]. All these algorithms in general follow one of the three well-known paradigms: divide-and-conquer [Dwyer 1987], sweep-line [Fortune 1987] and incremental insertion [Guibas et al. 1992]. On the other hand, algorithms for constructing the CDT can be grouped into two categories: (a) processing points and constraints simultaneously and (b) processing points and constraints separately. In category (a), Chew [1989] shows that the CDT can be built by using a divide and conquer approach to partition the problem into smaller subproblems within vertical strips. In category (b), since the CDT is a generalization of the DT with the notion of constraints [Lee and Lin 1986], we can first construct the DT of the given point set, then insert constraints one by one into it. Such an insertion can be done either by removing triangles pierced through by each constraint and re-triangulate the region due to the removal of these triangles, or by flipping some edges in a certain order until the constraint appears in the triangulation. Our approach of computing the CDT on the GPU lies in-between these two categories. We first construct a triangulation of the given point set, then insert all the constraints using edge flipping, followed by transforming the resulting triangulation into the CDT, also using edge flipping.

3 Computing the DT on the GPU

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

Phase 1. Digital Voronoi diagram construction. Map the input points into a grid and compute its digital VD. If more than one point is mapped to a same grid point, keep just one and treat the other as missing points.

Phase 2. Triangulation construction. Find all the digital Voronoi vertices to construct triangles for a triangulation. This triangulation is an approximation of the DT.

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

Phase 4. Missing points insertion. Insert all missing points to be a part of the triangulation.

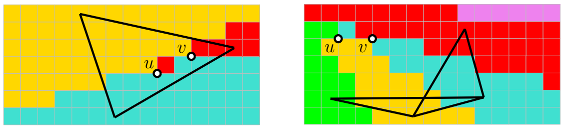
Phase 5. Edge flipping. Verify the empty circle property for each edge in the triangulation, and perform edge flipping if necessary.

Compared to the previous hybrid approach [Rong et al. 2008], the transformation (Phase 3 to Phase 5) in our algorithm 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 [1996a] in our computation. A list of triangles is stored in a pre-allocated array of size no more than $2|S|$, each one has the indices of up to three other triangles that are edge adjacent to it. Each vertex in S also has a linked list of triangles incident to it.

3.1 Phase 1: Digital Voronoi diagram construction

We first translate and then scale the points such that their 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 as a site for computing a digital VD. In case several points are mapped to a same grid point, only one among them is recorded, while the rest become *missing points* and will be handled later. Applying the Parallel Banding Algorithm (PBA) of Cao et al. [2010b], we can compute the mentioned digital VD on the GPU.

Cao et al. [2010a] show that by dualizing the output of the Standard flooding algorithm, one gets a valid triangulation of the points. The output of the Standard flooding is very close to that of PBA except that each region (of a same color) is connected. We adopt PBA as it is a much faster process than the Standard flooding, but the dual of its output may not be a geometrically valid triangulation. In particular, a region obtained by PBA can be disconnected and the dual of the digital VD thus can have duplicate and intersecting triangles; see Figure 2. We can quickly amend the output of PBA so that the claim in [Cao et al. 2010a] remains applicable. The detailed discussion on this modification can be found in Appendix A.



(a) (b)

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

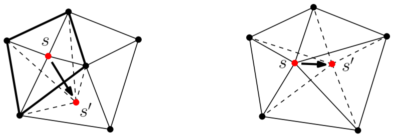
3.2 Phase 2: Triangulation construction

We dualize the result from the previous phase. A corner shared by up to four grid cells is incident to one to four different colors. For a corner with three (or four) colors, i.e., a *digital Voronoi vertex*, we add one (or two) triangle(s) into the triangulation.

During Phase 1, points are translated, scaled, and then slightly shifted from their original positions. We reverse this process 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 is not handled. Further discussion about this issue, which was neglected in the previous work [Rong et al. 2008] (that can result in a wrong output), can be found in the Appendix B.

3.3 Phase 3: Shifting

We say two points are neighbors when they are endpoints of an edge in the triangulation. Assuming the neighbors of s are static, shifting s may (*bad case*) or may not (*good case*) cause any intersection (Figure 3). The former happens when s moves across the boundary formed by its neighbors. As expected due to the very small shifting distance, majority of the cases in practice are the good cases.



(a) Bad case. (b) Good case.

Figure 3: *Shifting a point s to s' (a) may or (b) may not require some modifications to the triangulation.*

To achieve regularized work while shifting points in parallel, we separate the processing into two stages. During the first stage, we only shift a point if that does not require any fixing on the triangulation. To do so, we perform this stage in multiple iterations with all the points initially unmarked. In each iteration, each parallel thread in charge of an unmarked point s_i first checks if all its neighbors having indices smaller than i are marked. 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, then we shift it. Otherwise, we leave it for the second stage. After a point is processed, it is marked. Since most points would be processed and marked in the first few iterations, after each iteration we use compaction to remove marked points to speed up the later iterations.

During 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. For a parallel thread to delete one point, it first marks all triangles in its fan as deleted and then uses the ear-cutting method [Highnam 1982] to re-triangulate the resulting star-shape hole. Since the number of new triangles to be created is no more than the number of deleted triangles, we can use the deleted slots in the triangle list to store new triangles, with no racing memory access during parallel computation.

3.4 Phase 4: Missing points insertion

Missing points were identified in Phase 1 and Phase 3. We enclose S in a pre-defined regular polygonal boundary so that when we insert the missing points, they fall inside the triangulation. This boundary is removed at the end of this phase.

The insertion for each missing point p_i starts by identifying the triangle(s) in the triangulation that contains p_i, or has an edge passing through p_i. For a missing point obtained from Phase 1, we can start searching from a triangle incident to the point of S mapped to the same grid point as p_i; as for one from Phase 3, we can start searching 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 inserted, we delay the insertion of p_i to a later iteration. This searching is done in parallel with one thread handling one point.

To avoid concurrent modification of a triangle during the parallel insertion, this phase is also done in multiple iterations. In one iteration, each thread handling an insertion first uses the index of the point to be inserted to mark on those triangles to be modified. Then each thread checks the marks on the triangles and only performs the insertion if its marks are not overwritten by other threads. A global synchronization, which is fairly cheap for the GPU, is required to make sure that all the markings are done before any check is performed. This process is repeated until all missing points are inserted. The marking is done using an *atomic minimum* operation [NVIDIA 2011], which is readily available in the GPUs. This guarantees the termination of the algorithm, since in each iteration the missing point with the smallest index can always be inserted.

3.5 Phase 5: Edge flipping

We now verify the empty circle property for each edge in our triangulation in parallel. For an edge ab of the triangle abc, we only have to check if the point d of the adjacent triangle adb is inside the circumcircle of abc. If so, an edge flip is performed replacing abc and adb with the triangles 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 parallel thread to process one triangle, and we mark those that do not need any flipping, so we do not need to check it again in the next iteration. Note that when a pair of triangles is flipped, both of them need to be unmarked.

4 Computing the CDT on the GPU

To introduce constraints into the DT computation, we use the approach of computing 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.

The naïve approach of having one parallel thread to handle one constraint, deleting triangles that it pierces through and re-triangulating the created region is not ideal: each constraint can intersect a different number of triangles in $\mathcal{T}$, resulting in unbalanced workloads. Furthermore, two different threads handling two constraints may intersect some common triangles and the threads cannot proceed without some sort of locking, which is very costly 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 two constraints intersect some common triangles, we can still possibly flip some of these common triangles. 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 constraint), and perform edge flipping if necessary.

Step 3 is similar to Phase 5 of the DT algorithm, with some slight modification to avoid flipping constraints. Our proposed Step 2, with an *outer loop* and an *inner loop*, is given in Algorithm 1. The idea is to identify constraint-triangle intersections with the outer loop, and use edge flipping to remove them in the inner loop, all in parallel using multiple passes.

Algorithm 1 Inserting constraints into the triangulation

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

4.1 Outer loop: Find constraint-triangle intersections

For each triangle in the triangulation, we find the index of a 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 required in 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 GPU with one parallel thread processing one constraint.

4.2 Inner loop: Remove intersections

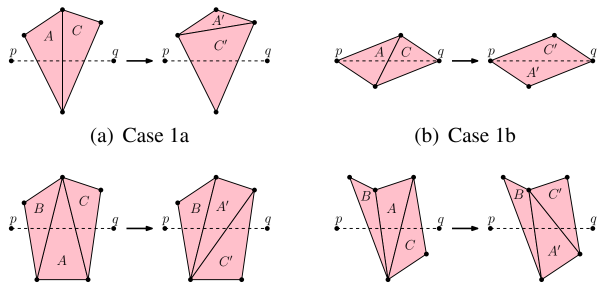
The inner loop of Algorithm 1 performs edge flipping to reduce the number of constraint-triangle intersections. Here, the parallelism is fine-grained with each thread processing a triangle. Consider a pair of triangles sharing an edge and both triangles are marked by the same constraint. Such a pair is classified as a *double intersection*, *single intersection* or *zero intersection*, respectively, if flipping it results in a new pair having two, one or zero intersections, respectively, with the constraint. If the flipping is not allowed as its underlying space is a concave quadrilateral, the pair is classified as *concave*.

Though it might seem reasonable to avoid flipping a double intersection case since it does not "improve" the situation and we may enter into some infinite loop, restricting flipping to only zero and single intersection may not be sufficient to remove all the intersections. We use a one-step look-ahead to overcome this dilemma. 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 4):

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



(a) Case 1a (b) Case 1b

(c) Case 2 (d) Case 3

Figure 4: *Flipping consideration of triangle pairs involving A. Constraint pq intersects triangles from left to right.*

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 2. In each iteration, we first identify triangle pairs and their cases. 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) if the case involves 3 triangles, with the index of A, using the atomic minimum operation. Lastly, we flip a pair of triangles only if all their marks remain. This is to prevent possible conflicts when updating the triangulation, and for the one-step look-ahead to be achieved. We also introduce extra weight into the label used in the marking to favor Case 1. For each step to be done in parallel, we assign one thread to process one triangle. As an optimization, we maintain a compact list of *active* triangles, i.e. those that still intersect their recorded constraints, after each iteration so that we do not have too many idle threads.

Algorithm 2 Processing of constraint-triangle intersections

```
repeat
  for each triangle $A$ intersecting a constraint do in parallel
    if $C$ is also marked by the same constraint then
      determine the case of $(A, C)$
    end if
  end for
  for each triangle $A$ intersecting a constraint do in parallel
    if $(A, C)$ is flippable then
      mark $A, C$ (and $B$ for Case 2) using atomic minimum
    end if
  end for
  for each triangle $A$ intersecting a constraint do in parallel
    if $A, C$ (and $B$ for Case 2) retain the same mark then
      flip $(A, C)$ and update the links between the new
      triangles and their neighbors
    end if
  end for
until no edge is flippable
```

In practice, the **repeat-until** loop of Algorithm 2 should be executed only a few times per each outer loop iteration instead of repeating until no edge is flippable. This is because as the algorithm progresses, there is a drastic reduction in the number of flippable cases, and the parallelism thus reduces. By switching to the outer loop after a few (say 5 to 10) iterations of inner loop, the algorithm can discover more flippable cases to improve the parallelism and thus improving performance without compromising the correctness of the algorithm proven in the next section.

4.3 Proof of correctness and complexity analysis

We show here that Algorithm 1 indeed 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 can always successfully insert a constraint into the triangulation.

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

If we would remove all triangles intersecting c_i, a polygonal hole is created with vertices $p_1, p_2, \ldots$ as its upper part and $q_1, q_2, \ldots$ as its lower part, excluding a and b; see Figure 5. Any polygon has an ear, so 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 in the chain. We exclude a and b themselves to be q_k. Let A be the previous triangle of C, 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. If $B = q_kp_{j-1}q_{k-1}$ then (B, A) is a double intersection; the union of triangles B, A, C is a convex polygon as needed in Case 2. If $B = q_kp_{j-1}p_{j-2}$ then (B, A) is a concave pair; because $p_{j-2}p_{j-1}q_{k+1}q_k$ is convex by the choice of q_k, triangles B, A, C fulfill Case 3. As long as there is one flippable triangle pair, the marking in the second **for** loop will successfully mark one for flipping, and flipping is indeed performed for each pass of the inner loop.

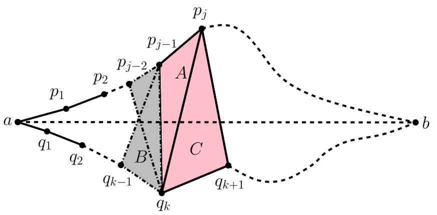


Figure 5: *Consideration when triangle pairs intersecting constraint $c_i = ab$ are either double intersection or concave.*

We next show that our inner loop does not continue forever. Let us assign to each triangle pair a value of 0, 1 and 2, respectively, according to its being zero/single intersection, double intersection and concave, respectively. Then, we have a base 3 number, N, to record the cases of the chain of triangles intersecting c_i. A flipping due to Case 1 deletes a digit in N, Case 2 turns 11 into 01, and Case 3 turns 21 into 11. In other words, each flipping decreases the value of N. Since N is finite, our algorithm clearly terminates, and a constraint is inserted as claimed. □

The above concludes that our proposed algorithm computes correctly the CDT. It also indirectly shows that no flip is wasteful with the following bound on the number of flips per constraint:

Claim 2. The total number of flipping performed by the inner loop to add 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 cannot be done more than k times too.

There are initially $O(k)$ concave pairs. A flipping due to Case 1 (or Case 2) can introduce at most two (or one) concave pair(s), 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. As such, Case 3 can be performed no more than $O(k^2)$ times. As a result, the total number of flipping is $O(k^2)$. □

5 Experimental 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 [1996b].

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 GPU-CDT program, we compare its running time, on both synthetic and real-world data (contour maps freely available at https://www.ga.gov.au/), with that of the most popular computational geometry softwares available, *Triangle* and CGAL version 3.9. As a side note, we can deduce indirectly that our work is superior to the prior work reported by Rong et al. [Rong et al. 2008] for the DT computation since a majority of their work is still performed sequentially. 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 between the two.

5.1 Synthetic Dataset

To generate synthetic data, we first randomly generate constraints of different lengths that do not intersect each other, then randomly generate points which do not lie on any constraint.

5.1.1 Comparison on DT results

For different number of points, our approach achieves 4 to 4.5 times speedup over CGAL; see Figure 6. Different grid sizes used for the

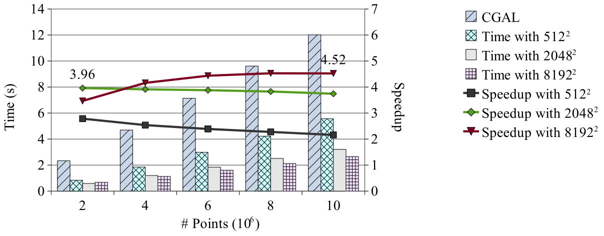
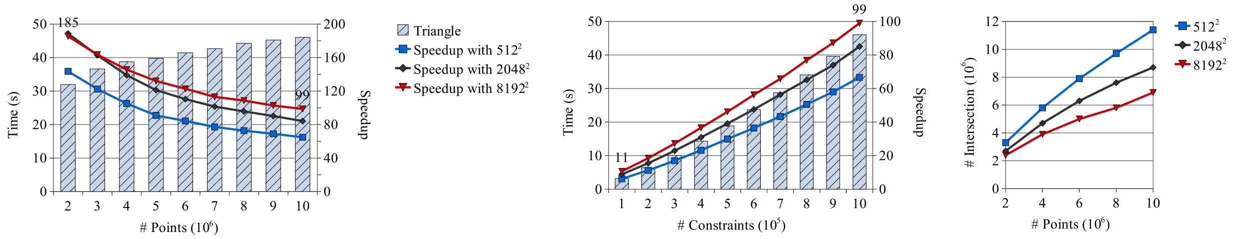


Figure 6: *Comparison on DT computation between GPU-CDT and CGAL. The running time of GPU-CDT on grid sizes 1024^2 and 4096^2 are omitted for clarity.*



(a) (b) (c)

Figure 7: *(a) Speedup over Triangle when computing the CDT with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints. (c) Total number of triangle-constraint intersections with different grid sizes.*

digital VD computation significantly affect the running time of our program. Generally, a larger grid gives a better approximation and less missing points, thus Phase 4 and Phase 5 run faster, although with some penalty on the digital VD computation time; see Figure 8(a). Here the running time of Phase 3 is also increased since more points (that can be mapped onto the grid) need to be shifted for larger grid. As a general guideline, given a larger set of input points, a larger grid is preferable.

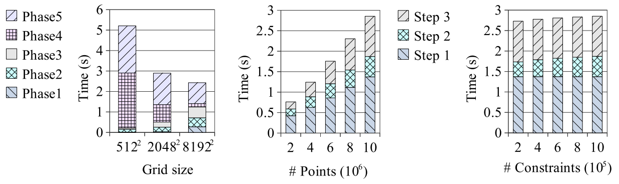
The same conclusion is also true when the input points are of a Gaussian distribution. For small grid, the speedup achieved is slightly lower due to more points being concentrated in the center of the grid and become missing points. The speedup increases quickly when using larger grid. However, this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

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 (also using an edge-flip method) on the DT of the point set. We compare the time for constraints insertion by subtracting the time for the DT computation from the time for the CDT computation on the same point set.

Figure 7(a) and 7(b) show the performance comparison of *Triangle* with GPU-CDT on different number of points and constraints, with different grid sizes. Clearly, the more constraints there are, the higher is the speedup we can achieve. This is because only a small part of our algorithm in inserting constraints is done with coarse-grained parallelism, while the majority of the processing is done with fine-grained parallelism. As such, 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 7(c)), possibly due to the fact that the triangulation produced by Step 1 is closer to the DT with grid size getting bigger.

Figure 8(b) and 8(c) shows the running time of different phases of GPU-CDT using 8192^2 grid size. Similar behavior is also observed for other grid sizes. The time for inserting constraints for our program occupies less than 20% of the total time. On the same datasets, *Triangle* spends most of its time inserting constraints. For example, given 10M points and 1M constraints, *Triangle* spends 62 seconds for constructing the CDT, in which 46 seconds are spent on constraints insertion. As such, when comparing the total running time of our program with that of *Triangle*, we achieve significant speedup, ranges from 10 to 45 times.



(a) (b) (c)

Figure 8: *Running time for different phases/steps for computing DT and CDT. (a) DT with 10M points where running time on grid sizes* 1024^2 *and* 4096^2 *are omitted for clarity. (b) CDT with 1M constraints and varying the number of points. (c) CDT with 10M points and varying the number of constraints.*

5.2 Real-world dataset

Figure 1(a) shows an example of the contour maps we used for our experiment and its CDT. The running times are presented in Table 1.

Example	# Points	# Constraints	Constraints insertion (sec)		Speedup
			Triangle	GPU-CDT	
a	1,177,332	1,176,943	0.665	0.046	14×
b	3,180,037	3,179,251	1.982	0.071	28×
c	4,461,519	4,460,506	2.526	0.097	26×
d	5,721,142	5,719,895	3.181	0.133	24×
e	8,569,881	8,568,121	4.755	0.245	19×
f	9,546,638	9,544,461	6.036	0.244	24×

Table 1: *Running time of contour dataset.*

GPU-CDT generally runs faster than *Triangle*. In these real-world data, most constraints are very short and do not intersect many triangles (if at all). Figure 9 shows the distribution of the number of intersections per constraint collected by our GPU-CDT for the Example f contour dataset (with about 10M points and 10M constraints) and a representative synthetic dataset of 10M points with 1M constraints. The maximum number of intersections is 51 for the

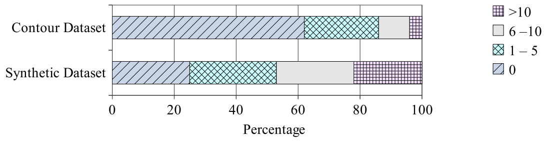


Figure 9: *The distribution of the number of intersections per constraint.*

contour dataset as compared to 7073 for the synthetic dataset.

For both cases, the total number of constraint-triangle intersections is around 6M. *Triangle* inserts constraints much slower when the constraints are long (one constraint intersects many triangles), taking 46 seconds for the synthetic dataset, while only 6 seconds for the contour dataset with mostly short constraints. 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 with similar parallelism as the contour dataset with all very short constraints. Our running time for both cases is roughly the same, achieving 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 [2006] 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 straight line segments is constructed. Third, the CDT to the edge map (which is actually a PSLG) is computed. Finally, adjacent *trixels* (triangles) are merged based on certain grouping filters, and the connected components of the trixel grouping graph yield polygons that represent the image. Figure 1(b) shows the result after computing the CDT of the edge map of the *Lena* image. 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 GPU-CDT can help speed up the computation. The constraints here are similar in nature to those in the contour datasets, so the performance of GPU-CDT here is similar to that for contour datasets.

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, capable of maximizing the parallelism on the GPU. That has been shown in our experiment with both synthetic and real-world data. We have shown that our implementation can achieve an order of magnitude better performance than the best CPU libraries available. One current limitation of our approach is that we construct the DT using the digital VD computed on a uniform grid. As a result, our algorithms is very fast when the input points are uniformly distributed, while performs less efficiently when the input points are in a highly skewed distribution. Nevertheless, we believe our approach is useful for many practical applications.

References

AURENHAMMER, F. 1991. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys 23*, 3, 345–405.

BOISSONNAT, J.-D. 1988. Shape reconstruction from planar cross sections. *Computer Vision, Graphics, and Image Processing 44*, 1, 1–29.

CAO, T.-T., EDELSBRUNNER, H., AND TAN, T.-S. 2010. Proof of correctness of the digital Delaunay triangulation algorithm. http://www.comp.nus.edu.sg/~tants/delaunay2DDownload_files/notes-30-april-2011.pdf.

CAO, T.-T., TANG, K., MOHAMED, A., AND TAN, T.-S. 2010. Parallel banding algorithm to compute exact distance transform with the GPU. In *I3D '10: Proc. ACM Symp. Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 83–90.

CGAL, 2011. CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

CHEW, L. P. 1989. Constrained Delaunay triangulations. *Algorithmica 4*, 97–108.

DWYER, R. 1987. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica 2*, 137–151.

FISCHER, I., AND GOTSMAN, C. 2006. Fast approximation of high-order voronoi diagrams and distance transforms on the GPU. *J. Graphics Tools 11*, 4, 39–60.

FORTUNE, S. 1987. A sweepline algorithm for Voronoi diagrams. *Algorithmica 2*, 153–174.

FORTUNE, S. 1997. *Handbook of discrete and computational geometry*. CRC Press, Inc., Boca Raton, FL, USA, ch. Voronoi diagrams and Delaunay triangulations, 377–388.

GUIBAS, L., KNUTH, D., AND SHARIR, M. 1992. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica 7*, 381–413.

HIGHNAM, P. T. 1982. The ears of a polygon. In *Information Processing Letters*, 196–198.

HOFF, III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH '99*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 277–286.

KALLMANN, M., BIERI, H., AND THALMANN, D. 2003. Fully dynamic constrained Delaunay triangulations. In *Geometric Modelling for Scientific Visualization*, G. Brunnett, B. Hamann, and H. Mueller, Eds. Springer-Verlag.

LEE, D., AND LIN, A. 1986. Generalized Delaunay triangulation for planar graphs. *Discrete and Computational Geometry 1*, 201–217.

NVIDIA, 2011. NVIDIA CUDA C Programming Guide, Version 4.0. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

PRASAD, L., AND SKOURIKHINE, A. N. 2006. Vectorized image segmentation via trixel agglomeration. *Pattern Recognition 39* (April), 501–514.

RONG, G., TAN, T.-S., CAO, T.-T., AND STEPHANUS. 2008. Computing two-dimensional Delaunay triangulation using graphics hardware. In *I3D '08: Proc. Symp. Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 89–97.

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

SHEWCHUK, J. R. 1996. Robust adaptive floating-point geometric predicates. ACM, New York, NY, USA, SoCG '96, 141–150.

SU, P., AND SCOT DRYSDALE, R. L. 1997. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry: Theory and Applications 7* (April), 361–385.

A Our Flooding Algorithm

Phase 1 of our proposed algorithm is to compute a digital VD of a collection of points, or called *seeds*, in a grid. Technically, we want and will prove in the following that the output of this phase is the same as that of the Standard flooding mentioned in [Cao et al. 2010a]. Then, we can apply their result to conclude that Phase 2 of our algorithm indeed produces a triangulation.

In our algorithm, we use the efficient Parallel Banding Algorithm (PBA) [Cao et al. 2010b] running on the GPU to start the coloring, which results in an Euclidean coloring. Each of the Voronoi region resulted has a connected component called *bulk* which is path-connected to its seed, and *debris* (if any) which are disconnected from the seed. Cao et al. show that bulks are subsets of the result of the Standard flooding. So, our challenge is to identify and recolor the debris to be the same as in the Standard flooding. Since there are very few debris in general, the recoloring is done using the CPU and then the result is sent back to the GPU to complete the coloring. Algorithm 3 is our proposed approach. We use Q to mean 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 is to remove and return the pair with minimum distance between the grid point and the seed of the 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).

Algorithm 3 Flooding Algorithm

```
Compute coloring with PBA, and identify all debris as uncolor.
$Q = \emptyset$
**for each** debris $A$ **do**
  $Q = Q \cup \{(A, i) \mid B \in N(A)$ having been colored $i$, and
                    $\|B - p_i\| \prec \|A - p_i\|\}$
**while** $Q \neq \emptyset$ **do**
  $(A, i) = \min(Q)$
  **if** $A$ is not colored **then**
    Color $A$ with color $i$
    $Q = Q \cup \{(B, i) \mid B \in N(A)$ and $\|A - p_i\| \prec \|B - p_i\|\}$
  **end if**
**end while**
```

Consider, on the contrary, the very first instance when Algorithm 3 colors a debris A with color r (inside the while-loop) whereas the Standard flooding produced grid point A with color $s \neq r$. There are two situations to consider:

CASE 1: $\|A - p_r\| \prec \|A - p_s\|$. From Algorithm 3, there exists a neighbor B of A colored by r earlier, and $\|B - p_r\| \prec \|A - p_r\|$. It follows that $\|B - p_r\| \prec \|A - p_r\| \prec \|A - p_s\|$. According to our choice of A, grid point B is colored by r in the Standard flooding. By the Ordered Coloring Lemma [Cao et al. 2010a], (A, r) must have been considered in the Standard flooding before (A, s) and A should thus have been colored by r then, 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. Part of this path has been colored the same (with s) in Algorithm 3 when we are about to color A. Let C be the grid point closest to p_s in that path not yet colored by Algorithm 3. With the previous grid point before C has been colored with s, (C, s) must have been added to Q in Algorithm 3. 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 the algorithm colors all grid points. This concludes our claim of correctness of our flooding algorithm.

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 x-axis; the discussion can be generalized to 2D with *scale* be the larger one calculated from both dimensions, 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 is to eventually shift all points in the grid back to their positions given in the input. To maintain as many good cases of shifting 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 our new coordinate of a point before shifting it (to negate the effect of the truncation to integer coordinate) to the original coordinate x. 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 $(\bar{x} \times \textit{scale} + \textit{translation})$. Let the number of bits used to represent the mantissas of the two constants *scale* and *translation* be pS and pT, respectively. Note that $\bar{x}$ is a non-negative integer with maximum value of $(m - 1)$ and thus needs $pM = (\log m)$ bits to represent. We keep $pT = p\text{Max}$.

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 round up can just increase *scale* by a little bit at its least significant bit 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: $\textit{translation} > \textit{range}$. Let 2^r be the largest term in the binary representation of *translation*. We round up all terms in the binary representation of *scale* that are smaller than $2^{r-(p\text{Max}-1)+pM}$. Because $\textit{range} = x_{\max} - x_{\min} \geq 2^{r-(p\text{Max}-1)}$, we have *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 round up does not increase more than twice the value of *scale*, and we thus still be able to spread out the mapping of points on the grid.