gCDT: A Highly Parallel GPU Algorithm for Large-Scale Constrained Delaunay Triangulation

PENG FAN, Zhejiang University, China
MIN TANG*, Zhejiang University, Zhejiang Sci-Tech University, China
RUOFENG TONG, Zhejiang University, China
LILI HE, Zhejiang Sci-Tech University, China
PENG DU, State Key Lab of CAD and CG, Zhejiang University, China
HAILONG LI, Shenzhen Poisson Software Co., Ltd., China

Fig. 1. **Letters benchmark.** For a scene containing approximately 900K vertices with numerous constraints of varying sizes, *gCDT* achieves an overall speedup of roughly 25X over the prior state-of-the-art GPU-based CDT algorithm *gDel2D* [Qi et al. 2013, 2019], while the constraint-processing component is nearly three orders of magnitude faster.

2D constrained Delaunay triangulation (CDT) is a key component in CAD, visualization, and scientific computing. We present a highly parallel GPU-based method capable of computing CDTs on inputs with millions of vertices, while efficiently and robustly enforcing arbitrary valid constraints. On benchmarks with complex constraints, our method typically delivers several-fold speedups over the prior state-of-the-art GPU approach and is roughly an order of magnitude faster than widely used CPU implementations. The efficiency stems from an improved parallel edge-flipping scheme coupled with a constraint-handling algorithm with provable time complexity, enabling stable performance even on adversarial and difficult configurations.

CCS Concepts: • **Computing methodologies** → **Modeling and simulation**; **Massively parallel algorithms**.

Additional Key Words and Phrases: 2D Constrained Delaunay Triangulation (CDT), Graphics Processing Units (GPUs), Parallel Edge-flipping Scheme

*Corresponding author, https://min-tang.github.io/home/gCDT/, This work was supported in part by the New Generation Artificial Intelligence-National Science and Technology Major Project (2025ZD0123901).

Authors' addresses: Peng Fan, Zhejiang University, China, fanpeng0103@zju.edu.cn; Min Tang, Zhejiang University, Zhejiang Sci-Tech University, China, tang_m@zju.edu.cn; Ruofeng Tong, Zhejiang University, China, trf@zju.edu.cn; Lili He, Zhejiang Sci-Tech University, China, llhe@zju.edu.cn; Peng Du, State Key Lab of CAD and CG, Zhejiang University, China, dp@zju.edu.cn; Hailong Li, Shenzhen Poisson Software Co., Ltd., China, lihailong@poissonsoft.com.

ACM XXXX-XXXX/2026/4-ART
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Peng Fan, Min Tang, Ruofeng Tong, Lili He, Peng Du, and Hailong Li. 2026. *gCDT*: A Highly Parallel GPU Algorithm for Large-Scale Constrained Delaunay Triangulation. 1, 1 (April 2026), 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Constrained Delaunay triangulation (CDT) [Lee and Lin 1986] is a fundamental primitive for planar meshing and planar straight-line graph (PSLG) processing: it produces a triangulation that preserves prescribed segments (constraints) while retaining Delaunay-type quality for downstream tasks such as interpolation, remeshing, and simulation [Chew 1989]. CDT appears throughout CAD sketch/profile processing, terrain and GIS breaklines, and multi-material interfaces, where boundaries and internal feature lines must be honored.

At modern scales, CDT is increasingly performance-critical yet must remain robust. Large point sets and dense constraint networks are triangulated repeatedly in iterative pipelines, and near-degenerate inputs (e.g., clustered points or near-collinearity) are common in real data. While optimal $O(n \log n)$ constructions exist in theory, practical performance is often dominated by constant factors, memory traffic, and the cost of robust geometric decisions. GPUs offer massive throughput and memory bandwidth, but CDT is difficult to map efficiently: the algorithm features irregular control flow, dynamic topology updates, and constraint enforcement that must avoid race conditions and invalid intermediate states.

We present *gCDT*, a highly parallel GPU algorithm for robust CDT on large PSLGs. The method reformulates key operations as bulk-synchronous, conflict-controlled primitives and enforces constraints using a strategy with predictable behavior, including challenging adversarial configurations. On an NVIDIA GeForce RTX 4090, *gCDT*

computes CDTs for inputs with up to 10 million vertices in about 0.1 seconds, typically achieving several-fold speedups over the prior state-of-the-art GPU approach and roughly an order-of-magnitude speedup over widely used CPU implementations.

Our contributions are:

- A parallel vertex-insertion stage that constructs a lightweight history tree for efficient point location;
- A parallel constraint enforcement stage based on load-balanced strip walking, ordered cavity retriangulation, and simultaneous handling of multiple constraints;
- A GPU-friendly edge-flipping phase that avoids long serial propagation while preserving concurrency.

2 RELATED WORK

CPU-based CDT. Robust CDT is widely available in mature CPU libraries such as Triangle [Shewchuk 1996b] and *CGAL* [Boissonnat et al. 2002; Fogel and Teillaud 2015], which are commonly used as baselines in CAD and scientific computing pipelines. Classic constrained-edge insertion and recovery strategies (e.g., Sloan [Sloan 1993]) are effective sequentially but introduce irregular dependencies and memory access patterns that hinder fine-grained parallelism. Recent work studies parallel Delaunay/CDT construction on CPUs and large-scale settings [Chen and Gotsman 2013; Lin et al. 2016; Rhodes 2018].

GPU Delaunay/CDT and constraint enforcement. Many GPU approaches target unconstrained Delaunay triangulation, often via iterative edge flipping that is amenable to massive parallelism [Navarro et al. 2011, 2012]. Extending these methods to CDT adds the need to insert and reliably recover constrained edges while maintaining a valid triangulation, which can become the dominant cost. Prior work explores this robustness–throughput trade-off for PSLGs and GPU-based CDT construction [Coll and Guerrieri 2017; Qi et al. 2013; Rong et al. 2008]. GPU-CDT [Qi et al. 2013] establishes an effective workflow that separates the main stages. Subsequently, inspired by 3D CDT algorithms, gDel2D [Cao et al. 2014] combined the point insertion and edge-flipping stages, achieving more than a twofold increase in efficiency. PCDT [Coll and Guerrieri 2017] further observes that constraint recovery is closely coupled with edge flipping and integrates these operations into a unified iterative loop. In contrast, *gCDT* treats constraint enforcement as a separate bulk-synchronous stage and uses load-balanced strip traversal and local retriangulation to avoid long flip-recovery chains. Elshakhs et al. provide a broader survey of Delaunay triangulation across paradigms and platforms [Elshakhs et al. 2024]. General-purpose systems [Jiang et al. 2022; Mahmoud et al. 2025] can serve as a foundation for GPU-based CDT. However, to optimize CDT-specific performance, we adopt a dedicated approach.

Robust predicates. Reliable orientation and incircle tests are essential for CDT, especially near degeneracies. Adaptive-precision predicates are a standard foundation for robustness guarantees [Shewchuk 1997]; Qi et al. present an efficient GPU implementation of this approach [Qi et al. 2019].

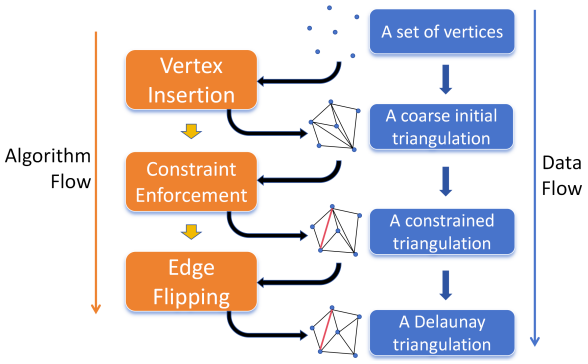


Fig. 2. **Algorithm workflow.** Overview of the algorithm workflow and the corresponding data transformations. The figure aligns each algorithmic stage with the data structures it updates.

3 ALGORITHM WORKFLOW

Figure 2 provides a roadmap of the method and the remainder of the paper. Given an input PSLG, we construct the final CDT via three stages:

- **Vertex insertion:** construct a coarse triangulation of all vertices (no constraints and no Delaunay guarantee yet).
- **Constraint enforcement:** enforce all constraint segments using a one-pass, highly parallel processing scheme.
- **Delaunay restoration:** iteratively flip non-Delaunay edges until the triangulation satisfies the Delaunay condition under the enforced constraints.

All stages rely on robust geometric decisions and exact arithmetic for key predicates; Section 4 summarizes the numerical foundations. A key design choice is to decouple vertex insertion from edge flipping, as is done in GPU-CDT (in contrast to Lawson-style coupled insertion used by gDel2D or PCDT), which improves GPU efficiency and simplifies synchronization; see Sections 5 and 7. Constraint enforcement is treated as a first-class parallel stage to avoid the high iteration counts and contention common in prior parallel CDT pipelines; details are in Section 6.

4 ARITHMETIC FOUNDATION

The construction and correctness of 2D constrained Delaunay triangulation (CDT) rely on two exact geometric predicates, $Orient2D(A, B, C)$ and $InCircle(A, B, C, D)$ (details provided in Appendix B). Adaptive-precision predicates are robust on CPUs [Shewchuk 1996a] but are harder to use efficiently on GPUs when many tests fall back to the exact path [Qi et al. 2019]. Since CDT with long straight constraints exhibits frequent near-collinear configurations, we use fixed-precision integer arithmetic, following the robust integer-arithmetic approach of Nehring-Wirxel et al. [2021]. In our fixed-precision pipeline, vertices are represented using integer coordinates. Integer arithmetic avoids round-off error, provided that intermediate computations do not overflow. For CDT predicates, we therefore bound the magnitude of input coordinates and use sufficiently wide integers for intermediate results. Because most

Algorithm 1 Initial triangulation by parallel vertex insertion.

Input: Vertex set V
Output: A valid triangulation T (not necessarily Delaunay)

Initialize T with a super-triangle that contains all vertices; set $v^{state} \leftarrow$ **Active** for all $v \in V$.
while $\exists v \in V$ s.t. $v^{state} \neq$ **Inserted do**
  Kernel 1 (Locate): for each $v \in V$ in parallel, if $v^{state} \neq$ **Inserted** then
    $(v^{tri}, v^{state}) \leftarrow \textsc{Relocate}(v, T)$
    $T[v^{tri}].choose \leftarrow v^{id}$ {last-writer-wins is sufficient}
  Kernel 2 (Split): for each triangle $t \in T$ in parallel, if $t.choose \neq \emptyset$ then
    $v \leftarrow V[t.choose]$
    $T \leftarrow \textsc{Split}(t, v)$
    if $v^{state} =$ **OnEdge then** $v^{state} \leftarrow$ **Active else** $v^{state} \leftarrow$ **Inserted**
    $t.choose \leftarrow \emptyset$
end while
return T

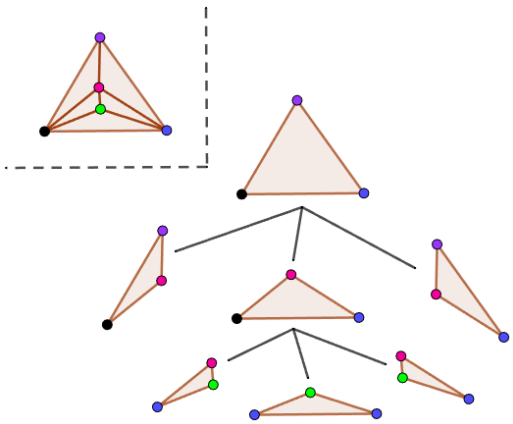


Fig. 3. **History tree.** When a triangle is split, we keep the original triangle as a parent and link it to its children. Identical vertices shared by multiple triangles are shown with the same color.

real-world inputs are represented in floating point, we first normalize and quantize them to integer coordinates. This quantization may introduce small geometric perturbations, which we discuss in Section 10.

Bit-width bounds. Assume input coordinates are quantized to signed b-bit integers after normalization. Then *Orient2D* fits in at most $2b$+3 bits, and *InCircle* fits in at most $4b$+4 bits (worst-case). We therefore evaluate both predicates with a fixed integer type; when supported, we use `int128` in device code [Conor Hoekstra 2022], otherwise we emulate wider integers with two 64-bit limbs. For 128-bit predicate evaluation, the worst-case bound $4b + 4 \leq 128$ gives $b \leq 31$, so we use $b = 31$ in our implementation.

Exact subdivision points. Constraint insertion (Section 6.2) creates n-division points on segment pq. We represent the k-th division point as an integer triple

$$(a, b, c) = \big((n-k)p_x + kq_x,\ (n-k)p_y + kq_y,\ n\big), \tag{1}$$

corresponding to $(a/c,\ b/c)$, and evaluate predicates by lifting all inputs to a common denominator, avoiding rounding.

5 VERTEX INSERTION

This section constructs an initial triangulation over the input vertices, serving as the substrate for subsequent Delaunay restoration and constraint enforcement (Section 6.2). Our insertion stage deliberately avoids edge flipping. This design has two benefits. First, it reduces the cost of point location during insertion: once a triangle is split, a point previously located in that triangle can only move into one of its (at most three) children, so point location can be updated with a constant number of predicate tests in expectation. Second, it preserves an insertion history tree (Figure 3) that we later reuse to accelerate constraint traversal.

Parallel workflow. Algorithm 1 follows a common bulk-synchronous GPU pattern [Cao et al. 2014; Qi et al. 2013]. Here, v^{tri} denotes the triangle currently containing vertex v, v^{id} is its array index, and v^{state} records its insertion state.

Each iteration consists of two kernels: (i) Locate assigns every not-yet-inserted vertex to a host triangle, and (ii) Split lets each triangle insert at most one vertex per iteration. We store in each triangle a 32-bit field $t.choose$ indicating the chosen vertex ID (or $\emptyset$). Multiple vertices may map to the same triangle in a given iteration; we do not require deterministic selection—any one of them suffices to ensure progress. Since $t.choose$ is a 4-byte write, the last-writer-wins behavior is acceptable in practice; at least one candidate will be recorded and inserted.

Vertex states and degeneracy. Each vertex v maintains a state in {**Active**, **OnEdge**, **Inserted**}. **Active** means v has not yet been inserted. **Inserted** means v is already a vertex of the current triangulation. **OnEdge** handles the degenerate case where v lies exactly on an existing edge (detected in Relocate using robust predicates). In Split, a regular vertex splits its host triangle into three triangles. For an **OnEdge** vertex, Split first splits only the selected incident triangle, creating two triangles, and then marks the vertex **Active**. The adjacent incident triangle is handled in a subsequent iteration, which completes the edge split without requiring both incident triangles to be claimed simultaneously.

Relocate and Split. Relocate performs point location starting from the triangle previously associated with v (cached as v^{tri}) and updates it after topological changes. Because insertion does not flip edges, the only local changes arise from triangle splits: An old triangle is replaced by its children, so v can only move among a constant-size neighborhood. This keeps relocation lightweight and makes it the dominant yet well-bounded cost per iteration.

History tree. We do not delete a triangle after splitting; instead, we retain it as a parent node and link it to its children, yielding a ternary tree over the insertion history (Figure 3). With a randomized or effectively shuffled insertion order, the expected depth is logarithmic under the standard randomized-insertion assumption, enabling

fast descent queries for later stages that need to map geometric entities (e.g., constraint segments) to intersected triangles in Section 6.2. Compared with more elaborate search structures such as the Delaunay tree [Boissonnat and Teillaud 1986] or related hierarchical schemes [Guibas et al. 1992], the history tree is intentionally simple, GPU-friendly, and sufficient for our downstream use.

6 CONSTRAINT ENFORCEMENT

6.1 Constraint enforcement workflow

Constraint enforcement is the core of our CDT pipeline. Instead of a flip-driven recovery that restores constraints via long chains of local edge flips, we use a retriangulate-and-stitch strategy, related to CPU-based approaches [Livesu et al. 2022; Shewchuk and Brown 2015], but reformulated here into bulk-synchronous GPU primitives. For each constraint, we (i) collect triangles intersected by the segment, (ii) remove the intersected strip to form two polygonal regions, and (iii) retriangulate them with the constraint on the boundary. This replaces irregular flip sequences with bulk-synchronous steps that map well to GPUs.

Sections 6.2–6.5 implement this strategy as four components: Section 6.2 finds all intersected triangles in parallel and addresses the key bottleneck in prior GPU methods - load imbalance caused by long constraints. Sections 6.3 and 6.4 describe polygon and mesh retriangulation: we first explain the single-constraint case to motivate the triangulation rule, and then generalize to the practical case where triangles may be intersected by multiple constraints. Finally, Section 6.5 analyzes and repairs the rare failure mode introduced by conflict resolution (holes), yielding a robust end-to-end procedure.

6.2 Find All Intersected Triangles

Serially tracing the triangle strip intersected by a constraint is standard: after locating the triangle containing one endpoint, one walks across adjacent triangles by segment–edge intersection until reaching the other endpoint. A direct GPU parallelization that assigns one thread per constraint, as in [Qi et al. 2013], implicitly assumes that constraints have comparable length. In practice, constraints can be arbitrarily long, and a single thread may traverse orders of magnitude more triangles than others, causing severe load imbalance.

We mitigate this by segmenting long constraints and processing the resulting segments independently. Using the history tree from Section 5, we can locate an arbitrary point in $O(\log n)$ expected time by descending the tree. Therefore, we split each constraint into multiple shorter segments, locate the starting triangle of each segment independently, and then apply the standard serial strip walk to each segment in parallel. To distribute work, we allocate threads from a fixed pool proportionally to constraint length, guaranteeing at least one thread per constraint. Figure 4 illustrates the idea: long constraints receive more threads and thus more segments, equalizing per-thread traversal cost. The split points in Figure 4 are used only for logical segmentation; they are not inserted as additional vertices into the triangulation. This strategy is simple, avoids global acceleration structures, and is effective in practice.

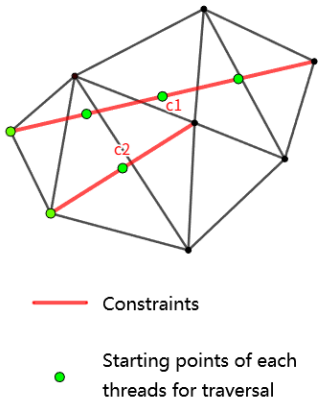


Fig. 4. **Load-balanced triangle intersection for constraints.** Each constraint is uniformly split into segments; segments are assigned to threads, and longer constraints receive proportionally more threads.

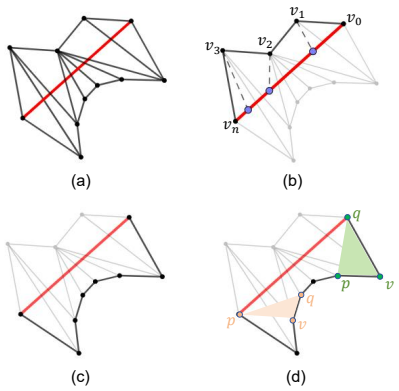


Fig. 5. **Constraint enforcement examples.** (a) A constraint intersects a contiguous strip of triangles. (b) One induced polygon and a feasible mapping to points on the constraint for visibility arguments. (c) The other polygon illustrates a worst-case scenario for parallel ear clipping. (d) Two examples of distance-ordered triangulation.

Although one could treat this as a generic 2D segment–triangle intersection problem and build spatial hashes or BVHs, those alternatives introduce additional construction and traversal overhead and typically produce unordered hits that require sorting/merging before polygon extraction. In contrast, strip-walking outputs intersected triangles naturally in traversal order, which directly supports the subsequent polygon construction.

6.3 Parallel Polygon Retriangulation

For exposition, we assume that each triangle intersects at most one constraint. Under this assumption, a constraint intersects a strip of triangles and splits the region into two simple polygons (Figure 5).

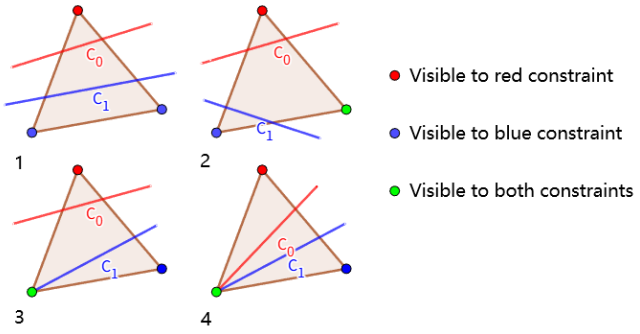


Fig. 6. **Constraint intersection cases.** Four topological cases of two constraints intersecting a triangle, and the corresponding vertex-visibility relations.

The task reduces to retriangulating each polygon while keeping the constraint segment as part of the boundary. An ear-clipping method runs in linear time for this class of polygons [Livesu et al. 2022], but its parallel efficiency is highly shape dependent. Parallel ear clipping can remove only locally convex ears in each iteration; in the worst case, only two vertices can be processed per iteration (Figure 5(c)), causing the parallel algorithm to degenerate into nearly serial execution. Because polygons induced by long constraints can be large and geometrically irregular, we require a method with predictable parallel depth.

Distance-ordered triangulation. The induced polygons are closely related to monotone polygons, enabling a direct adaptation of the parallel triangulation scheme in [Wagener 1988]. Let the polygon vertices be $v_0, v_1, \ldots, v_n$ in order, where (v_0, v_n) is the constraint. Let $\tilde{d}_i$ denote the signed perpendicular (unnormalized) distance of v_i to the supporting line of (v_0, v_n):

$$\tilde{d}_i = (v_n - v_0) \times (v_i - v_0), \tag{2}$$

where $\times$ is the 2D scalar cross product. We have $\tilde{d}_0 = \tilde{d}_n = 0$ and $\tilde{d}_i \neq 0$ for interior vertices in the non-degenerate case. Since only relative ordering matters, the normalization by $\|v_n - v_0\|$ can be omitted. When implemented using the exact integer arithmetic from Section 4, $\tilde{d}_i$ is computed without numerical error.

For each $i \in \{1, \ldots, n-1\}$, define

$$\begin{aligned} p_i &= \max\{\, j \mid 0 \le j < i,\ \tilde{d}_j < \tilde{d}_i \,\}, \\ q_i &= \min\{\, j \mid i < j \le n,\ \tilde{d}_j \le \tilde{d}_i \,\}. \end{aligned} \tag{3}$$

Then the set of triangles $\triangle(v_{p_i}, v_i, v_{q_i})$ forms a valid triangulation of the polygon (Figure 5(d)) [Wagener 1988]. Equivalently, we output edges (v_i, v_{p_i}) and (v_i, v_{q_i}) for all i and include the boundary edge (v_0, v_n).

Correctness. A proof of correctness for the distance-ordered triangulation is provided in Appendix A.1.

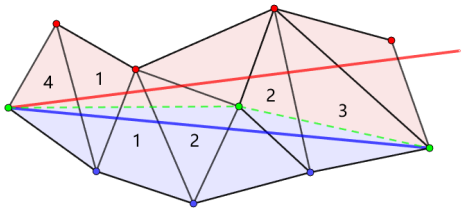


Fig. 7. **Conflicting constraints.** A conflicting pair of constraints intersecting a strip of triangles. The first and last triangles fall into one of the boundary cases (3 and 4) in Figure 6; interior triangles are of the shared-intersection case (1 and 2).

6.4 Parallel Mesh Retriangulation

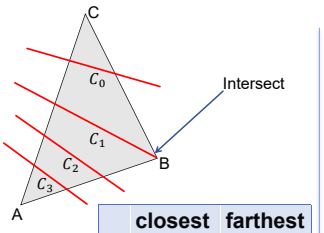
We now remove the simplifying assumption from Section 6.3 and handle the general case where a triangle can be intersected by multiple constraints. If we retriangulate each constraint-induced polygon independently, overlapping polygons would cause triangles to be generated multiple times, producing an invalid mesh. A naive workaround is to process only a maximal independent set of non-conflicting constraints per round. However, this can degenerate to $O(m)$ rounds for m constraints and offers little parallelism.

Our approach resolves conflicts within a single pass by ensuring that polygonal regions associated with different constraints become non-overlapping. The key operation is visibility pruning: when multiple constraints intersect the same triangle strip, we remove from each polygon the vertices that are not directly visible to that constraint because they lie "behind" another constraint. Importantly, we remove vertices rather than performing full polygon clipping. When two constraints intersect a triangle, symmetry yields four distinct configurations (Figure 6). If two constraints conflict, they intersect a contiguous strip of triangles (Figure 7); pruning invisible vertices is equivalent to clipping along the segment connecting the mutually visible intersection points in the boundary cases, eliminating the overlap between the two polygons. This produces disjoint retriangulation regions that can be processed concurrently.

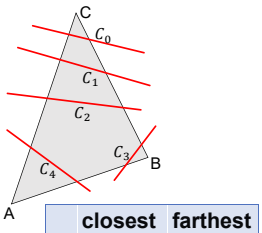
Visibility pruning can, in rare cases, create holes; we analyze the necessary condition and repair strategy in Section 6.5.

Implementation. From Section 6.2 we already have all constraint–triangle intersections. A direct implementation would test the visibility of each candidate polygon vertex by iterating over all constraints intersecting the triangle, which is expensive. We instead precompute, for each triangle vertex, a small summary that enables $O(1)$ visibility decisions during polygon construction.

For each triangle and each of its three vertices, we find up to two constraints: the closest and farthest constraints relative to that vertex (some may not exist). The computation considers only constraints that do not intersect the edge opposite the vertex. Among these, the closest constraint is the unique one visible to the vertex, while the farthest is the unique one occluded by all others (Figure 8). We additionally handle the special case where a constraint passes through the other two vertices exactly: such constraints must be considered when determining the closest constraint, but can be



	closest	farthest
A	C_3	C_2
B	-	-
C	C_0	C_0



	closest	farthest
A	C_4	C_4
B	C_3	C_3
C	C_0	C_2

Fig. 8. **Nearest vs. farthest constraints.** Two examples illustrating "closest" and "farthest" constraints relative to triangle vertices.

ignored for the farthest constraint (hence farthest may be undefined). This preprocessing is a parallel reduction over constraints incident to each triangle and runs in logarithmic time with respect to that local incidence count. Afterwards, the visibility of a vertex to a candidate constraint can be evaluated in constant time using the precomputed closest/farthest tags and a small number of local tests.

6.5 Filling Holes

We now address the failure mode introduced by visibility pruning: the creation of holes. Constraint enforcement should preserve the validity of the triangulation; in particular, the local retriangulation should cover exactly the region removed by intersected triangles. Because all triangles produced by our retriangulation are valid, holes can be detected by checking for missing coverage, equivalently by monitoring local triangle counts.

A convenient local accounting is as follows. For each new edge introduced that crosses a constraint, the number of triangles decreases by one; for each vertex that becomes visible to a constraint (excluding constraint endpoints), the number of triangles increases by one. Holes can occur only in the specific configuration where, in a triangle affected by multiple constraints, each of its three vertices is visible to a different constraint. For example, in Figure 8 (right), constraint C_4 does not increase the number of intersected edges (they were already intersected by other constraints), but it reduces the number of vertices visible to C_2 and C_3 by taking exclusive visibility of A. This reduces the number of triangles that should be generated by one, leaving a hole. All other configurations preserve the expected local count and therefore cannot produce holes.

Hole shape. All holes produced by our procedure are convex polygons; the proof is given in Appendix A.2.

Detecting holes and triangulating convex polygons are both straightforward in parallel, so we omit details. In practice holes are rare on real datasets and the repair overhead is negligible.

7 PARALLEL EDGE FLIPPING

7.1 Classical Algorithm Workflow

A widely used GPU edge-flipping scheme proceeds in bulk-synchronous rounds. Each round has two kernels. In the first kernel, every edge is tested in parallel using the InCircle predicate; non-Delaunay edges compete to "claim" their two incident faces by writing an identifier (often the edge ID or a random priority) into per-face records using atomic operations (e.g., `atomicMin`/`atomicMax`). In the second kernel, each edge re-reads the two face records. The edge is flipped only if it is the winner in both incident faces, which guarantees that no two concurrent flips share a face and thus preserves mesh validity. These rounds repeat until no edge violates the Delaunay criterion.

In our pipeline, vertex insertion is decoupled from edge flipping. This simplifies insertion and enables the data structure used in Section 6.2, but it means that the triangulation can be farther from Delaunay, increasing the number of required flips. This motivates an aggressive yet safe GPU strategy for the flipping stage.

Algorithm 2 Edge flipping propagation (one thread).

Input: A seed edge e_0, initialize per-face flags for this pass

```
FlipEdge(e_0)
Let f_0, f_1 be the two incident faces of e_0
Initialize a small local worklist S ← {e | e ∈ f_0, f_1, e ≠ e_0}
while S is not empty do
    Pop an edge e' from S
    if InCircle(e') then
        Let f' be the unlocked one of the two incident faces of e'
        if TryLock(f') then
            FlipEdge(e')
            S ← {e | e ∈ f', e ≠ e'} (bounded capacity)
        end if
    end if
end while
```

7.2 Improvement: Per-Thread Flip Propagation

Two empirical observations guide our optimization. First, despite the arithmetic cost of InCircle, runtime is dominated by memory traffic: repeatedly scanning the entire edge list and updating face records incurs substantial global reads/writes. Second, most flips occur in the first few rounds; later rounds process only a small residual set of edges, yet still pay the fixed overhead of launching kernels and sweeping large buffers.

We therefore aggregate work by allowing a thread that successfully flips an edge to continue flipping additional nearby edges in the same local region. This reduces (i) global passes over all edges, and (ii) the number of bulk-synchronous rounds, while keeping the same safety guarantee that no two concurrent flips operate on the same face.

We now describe Algorithm 2. Before propagation begins, we identify a set of mutually non-interfering seed edges using the same face-claiming mechanism as the classical workflow and initialize per-face atomic flags. The faces adjacent to seed edges are marked as locked, while all other faces are initialized as unlocked. A flag value of 1 denotes a locked face, and 0 denotes an unlocked face. Suppose a thread is assigned a seed edge E_{ab} shared by triangles T_a and T_b. After flipping E_{ab}, the mesh connectivity changes locally and may create new non-Delaunay candidates among adjacent edges, for example, an edge incident to T_b and T_c. Unlike the classical method,

which would terminate the thread and defer these candidates to later global rounds, our thread immediately tests neighboring edges and flips those that are eligible. To avoid conflicts, we use per-face atomic flags: an edge flip is performed only if the thread successfully locks the unlocked incident face. Since the other incident face has already been locked either during initialization or by the current thread, TryLock can be implemented using a simple atomic operation. This ensures that concurrent threads never flip edges that share a face, exactly as in the classical face-claiming approach, but now applied inside a local propagation loop.

Novelty and relation to prior GPU schemes. Classical GPU edge flipping treats each flip as an isolated operation selected by a global competition per round. Our method performs region-growing flip propagation: each thread executes a short sequence of dependent flips within a face-disjoint local region, amortizing memory traffic and kernel-launch overhead. Conceptually, the unit of work is no longer a single edge but a small flip chain confined by face locks.

Bounded local worklists. A practical challenge is implementing the neighbor exploration efficiently. Maintaining an unbounded set S is infeasible on the GPU: in the worst case, a single thread could touch a large fraction of the mesh, requiring dynamic allocation and unpredictable execution time. We therefore use a fixed-capacity local worklist per thread and discard overflow. In practice, we found that a small capacity (e.g., 8) captures the vast majority of profitable propagation opportunities. Moreover, near-optimal performance is achieved as long as the exploration reliably includes the edges incident to the two faces created/modified by the seed flip (i.e., the immediate 1-ring around the flipped edge); deeper propagation yields diminishing returns.

8 OPTIMIZATIONS

We present two optimizations that improve performance while preserving decoupling.

Upsampling (coarse-to-fine seeding). Decoupling vertex insertion from Delaunay edge flipping improves parallelism and enables the insertion history tree, but it can also produce an initial triangulation that is far from Delaunay for challenging point distributions (e.g., clustered or highly anisotropic samples). A poor initial mesh increases the number of non-Delaunay edges and the propagation distance of flips, inflating the cost of the subsequent flipping stage.

To mitigate this effect, we adopt a coarse-to-fine strategy. We first uniformly sample a small subset of vertices, compute a Delaunay triangulation on this subset, and then insert the remaining vertices into this triangulation using our parallel insertion procedure. This "seed" triangulation provides a higher-quality starting point, reducing the number of flips required later, while maintaining the structural advantages of decoupling. Empirically, we find that even a modest sample fraction yields a noticeable reduction in the total flip workload.

Pre-sorting for locality. Improving spatial locality is essential on GPUs because the edge-flipping stage is predominantly memory-bound. We therefore pre-sort vertices by Morton code [Lauterbach et al. 2009], a standard technique that increases cache coherence and improves the locality of neighborhood access during point location and insertion.

In addition, we apply the same idea to connectivity data by reordering triangles according to the Morton order of their centroids. Although edge flips modify local connectivity and can move triangles between neighborhoods over time, the computation remains largely local, and the initial ordering still improves cache behavior and memory coalescing for a substantial portion of the execution. In practice, this simple preprocessing consistently reduces memory-traffic overhead in the flipping kernels.

9 IMPLEMENTATION AND RESULTS

9.1 Implementation

We implemented *gCDT* in CUDA and evaluated it on an NVIDIA GeForce RTX 4090 (24 GB) using CUDA Toolkit 12.9. All GPU baselines were compiled and executed in the same environment. CPU baselines were evaluated on a desktop PC with an Intel Core i9–13900K and 32 GB RAM.

All inputs are sanitized by a lightweight preprocessing step (e.g., removing duplicate vertices and invalid/redundant constraints). Following common practice, we report runtimes excluding preprocessing, and apply the same convention to the compared methods.

We evaluate on two synthetic suites (unconstrained and constrained) and three vectorized-image datasets:

- **Synthetic:** Four benchmarks with 1M–9M vertices under four point distributions and no constraints (Figure 9).
- **Synthetic-cons:** Six benchmarks with 1M uniformly distributed vertices and 150K constraints; cases vary the constraint-length distribution from short segments to scene-spanning constraints (Figure 10).
- **Leafs:** Vectorized image with many short constraints and an approximately uniform spatial distribution (top of Figure 11).
- **Airport:** Vectorized image with a small number of long constraints and a non-uniform spatial distribution (bottom of Figure 11).
- **Letters:** Vectorized "SIGGRAPH" with multiple long constraints; includes a larger instance (Letters 910K) to highlight the challenging regime (Figure 1).

Baselines and numeric settings. Our primary GPU baseline is gDel2D [Cao et al. 2014; Qi et al. 2013, 2019], a widely used and actively maintained GPU-based CDT implementation. We also report results for PCDT [Coll and Guerrieri 2017] when it completes successfully. For CPU comparisons, we use CGAL as a canonical robust reference implementation. Unless stated otherwise, *gCDT* uses the fixed-precision `int128` predicates from Section 4, while GPU baselines use floating-point arithmetic with exact predicates [Qi et al. 2019].

9.2 Results

Synthetic (Delaunay only). On unconstrained triangulation, *gCDT* consistently outperforms gDel2D on uniform and moderately perturbed distributions, reflecting the benefits of our flip-stage optimization and locality improvements (Sections 7 and 8). As inputs become

more adversarial, the gap narrows, consistent with the reduced parallelism available to flip-based restoration. In this grid-structured setting, predicate computation becomes a dominant factor, so the performance gap between CGAL and GPU-based methods narrows relative to the other benchmarks. In contrast, *gCDT* benefits from its fixed-precision integer predicate pipeline and achieves a larger speedup over other GPU-based methods.

Synthetic-cons (constraints). With constraints, *gCDT* delivers large and stable gains, especially when long constraints are present. While prior GPU methods are sensitive to constraint length due to load imbalance and long recovery chains, our segment-wise traversal and retriangulation-based enforcement keeps runtime largely insensitive to constraint scale (Figure 13). PCDT does not complete the hardest constrained case in single precision in our environment; we report its double-precision result where applicable. Although robust CPU libraries remain strong baselines for complex constraint configurations, the throughput advantage of GPUs allows *gCDT* to remain faster than CGAL overall in these tests.

Real-world datasets. On vectorized-image datasets, *gCDT* provides strong end-to-end speedups and remains robust on challenging instances with long, non-uniform constraints. On the Letters benchmark, constraint enforcement becomes negligible in our pipeline; the overall runtime is then dominated by Delaunay restoration, while gDel2D remains bottlenecked by constraint enforcement. In the constraint module alone, *gCDT* is nearly three orders of magnitude faster than gDel2D on this dataset. CGAL also performs competitively on several benchmarks, surpassing the other GPU baselines in some cases. This is mainly because the Airport and Letters benchmarks have vertex and constraint distributions that are unfavorable to flip-based GPU methods, leading to more Delaunay-restoration iterations. Nevertheless, *gCDT* remains faster than CGAL on these benchmarks.

Overall, gCDT is most advantageous over gDel2D and PCDT when constraints are long or densely distributed, because segment-wise traversal and local retriangulation avoid long recovery chains. Its advantage over CGAL becomes most pronounced at larger input scales, where GPU throughput offsets synchronization overhead; the gap narrows when Delaunay restoration dominates the runtime.

Breakdown. We report a phase breakdown over representative benchmarks (Figure 14). "Insert" includes coarse-to-fine seeding; "Constraint" includes intersected-triangle discovery, retriangulation, and hole repair; "Flip" is Delaunay restoration. Constraint time correlates primarily with constraint geometry (especially length), whereas insertion and flipping are more sensitive to vertex distribution.

Ablation study. To evaluate the contribution of the edge-flipping optimization in Section 7, we conduct ablation experiments on benchmarks where flipping accounts for a large fraction of the total runtime (Figure 15). We compare three variants: *gCDT*-1 disables vertex sorting, *gCDT*-2 disables face sorting, and *gCDT*-3 disables the flip-propagation optimization described in Section 7.2. Face sorting is naturally enabled by our workflow and is evaluated as a separate locality optimization. On the Airport benchmark, vertices are already nearly ordered; therefore, additional vertex sorting provides only a marginal benefit and can slightly increase overhead. Overall, the proposed edge-flip propagation consistently reduces flipping time, and its benefit becomes more pronounced as scene complexity increases.

10 CONCLUSION AND LIMITATIONS

We presented *gCDT*, a GPU algorithm for large-scale 2D constrained Delaunay triangulation. By decoupling vertex insertion, constraint enforcement, and Delaunay restoration, *gCDT* converts an irregular CDT pipeline into three GPU-friendly phases that can be optimized independently. Our constraint module enforces arbitrary valid segments by load-balanced, segment-wise strip traversal and local retriangulation, avoiding long flip chains. For Delaunay restoration, we propose flip propagation that lets a thread perform a short, face-disjoint sequence of flips, reducing global rounds and amortizing memory traffic. Finally, fixed-precision integer predicates provide deterministic `Orient2D`/`InCircle` decisions on the GPU for inputs that fit within the prescribed fixed-precision bounds.

Experiments on synthetic and real planar straight-line graphs show substantial speedups over prior GPU and CPU implementations while maintaining stable performance under complex constraint networks. On an NVIDIA GeForce RTX 4090, *gCDT* triangulates inputs with up to 10 million vertices in about 0.1 seconds (excluding preprocessing).

Limitations. Our fixed-precision pipeline requires mapping coordinates to an integer grid; for floating-point inputs, quantization can merge nearby vertices and change near-degenerate configurations, producing results that may differ from exact-real implementations. For applications with stringent precision requirements, additional preprocessing may be needed to validate the quantized input, and postprocessing may be required to verify consistency with the original floating-point geometry. In such cases, a floating-point or adaptive-precision variant may be preferable. Moreover, as in other flip-based parallel restorations, adversarial distributions can reduce independent flips and increase iteration counts.

Future work. Improving the quality of the seed triangulation (beyond our simple coarse-to-fine sampling) could further reduce flip workload and improve worst-case behavior. We also plan to integrate *gCDT* into GPU pipelines for planar arrangements and Boolean operations, where CDT-like retriangulation is a key subroutine [Guo and Fu 2024; Levy 2025].

REFERENCES

Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. 2002. Triangulations in CGAL. Computational Geometry 22, 1–3 (2002), 5–19. https://doi.org/10.1016/S0925-7721(01)00054-2

J D Boissonnat and M Teillaud. 1986. The hierarchical representation of objects: the Delaunay tree. In Proceedings of the Second Annual Symposium on Computational Geometry (Yorktown Heights, New York, USA) (SCG '86). Association for Computing Machinery, New York, NY, USA, 260–268. https://doi.org/10.1145/10515.10543

Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, and Tiow Seng Tan. 2014. A GPU accelerated algorithm for 3D Delaunay triangulation. Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (2014). https://api.semanticscholar.org/CorpusID:3347902

Renjie Chen and Craig Gotsman. 2013. Localizing the Delaunay Triangulation and Its Parallel Implementation. Springer Berlin Heidelberg, Berlin, Heidelberg, 39–55. https://doi.org/10.1007/978-3-642-41905-8_4

L. Paul Chew. 1989. Constrained Delaunay Triangulations. Algorithmica 4, 1 (1989), 97–108. https://doi.org/10.1007/BF01553881

Narcís Coll and Marité Guerrieri. 2017. Parallel constrained Delaunay triangulation on the GPU. International Journal of Geographical Information Science 31, 7 (2017), 1467–1484. https://doi.org/10.1080/13658816.2017.1300804

Mark Harris Conor Hoekstra, Kuhu Shukla. 2022. Implementing High-Precision Decimal Arithmetic with CUDA int128. https://developer.nvidia.com/blog/implementing-high-precision-decimal-arithmetic-with-cuda-int128/

Yahia S Elshakhs, Kyriakos M Deliparaschos, Themistoklis Charalambous, Gabriele Oliva, and Argyrios Zolotas. 2024. A comprehensive survey on Delaunay triangulation: applications, algorithms, and implementations over CPUs, GPUs, and FPGAs. IEEE Access 12 (2024), 12562–12585.

Efi Fogel and Monique Teillaud. 2015. The computational geometry algorithms library CGAL. ACM Communications in Computer Algebra 49, 1 (2015), 10–12. https://doi.org/10.1145/2768577.2768579

Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. 1992. Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica 7, 1 (01 Jun 1992), 381–413. https://doi.org/10.1007/BF01758770

Jianzhi Guo and Xiao-Ming Fu. 2024. Exact and Efficient Intersection Resolution for Mesh Arrangements. ACM Transactions on Graphics (TOG) 43 (2024), 1 – 14. https://api.semanticscholar.org/CorpusID:274264568

Zhongshi Jiang, Jiacheng Dai, Yixin Hu, Yunfan Zhou, Jeremie Dumas, Qingnan Zhou, Gurkirat Singh Bajwa, Denis Zorin, Daniele Panozzo, and Teseo Schneider. 2022. Declarative Specification for Unstructured Mesh Editing Algorithms. ACM Trans. Graph. 41, 6, Article 251 (Nov. 2022), 14 pages. https://doi.org/10.1145/3550454.3555513

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics (Paris, France) (EGGH-HPG'12). Eurographics Association, Goslar, DEU, 33–37.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David P. Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. Computer Graphics Forum 28, 2 (2009), 375–384. https://doi.org/10.1111/j.1467-8659.2009.01377.x

Der-Tsai Lee and Arthur K. Lin. 1986. Generalized Delaunay Triangulations for Planar Graphs. Discrete & Computational Geometry 1 (1986), 201–217.

Bruno Levy. 2025. Exact Predicates, Exact Constructions and Combinatorics for Mesh CSG. ACM Trans. Graph. 44, 5, Article 167 (July 2025), 27 pages. https://doi.org/10.1145/3744642

Jiaxiang Lin, Riqing Chen, Changcai Yang, Zhaogang Shu, Changying Wang, Yaohai Lin, and Liping Wu. 2016. Distributed and Parallel Delaunay Triangulation Construction with Balanced Binary-tree Model in Cloud . In 2016 15th International Symposium on Parallel and Distributed Computing (ISPDC). IEEE Computer Society, Los Alamitos, CA, USA, 107–113. https://doi.org/10.1109/ISPDC.2016.79

Marco Livesu, Gianmarco Cherchi, Riccardo Scateni, and Marco Attene. 2022. Deterministic Linear Time Constrained Triangulation Using Simplified Earcut. IEEE Transactions on Visualization and Computer Graphics 28, 12 (2022), 5172–5177. https://doi.org/10.1109/TVCG.2021.3070046

Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2025. Dynamic Mesh Processing on the GPU. ACM Trans. Graph. 44, 4, Article 136 (July 2025), 19 pages. https://doi.org/10.1145/3731162

Cristobal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. 2011. A parallel GPU-based algorithm for Delaunay edge-flips. In Proceedings of the 27th European Workshop on Computational Geometry (EuroCG). Morschach, Switzerland. https://eurocg11.inf.ethz.ch/abstracts/35.pdf Extended abstract.

Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. 2012. A Quasi-Parallel GPU-Based Algorithm for Delaunay Edge-Flips. https://api.semanticscholar.org/CorpusID:18299299

Julius Nehring-Wirxel, Philip Trettner, and Leif Kobbelt. 2021. Fast Exact Booleans for Iterated CSG using Octree-Embedded BSPs. Computer-Aided Design 135 (2021), 103015. https://doi.org/10.1016/j.cad.2021.103015

Meng Qi, Thanh-Tung Cao, and Tiow Seng Tan. 2013. Computing 2D Constrained Delaunay Triangulation Using the GPU. IEEE Transactions on Visualization and Computer Graphics 19, 5 (2013), 736–748. https://doi.org/10.1109/TVCG.2012.307

Meng Qi, Ke Yan, and Yuanjie Zheng. 2019. GPredicates: GPU Implementation of Robust and Adaptive Floating-Point Predicates for Computational Geometry. IEEE Access 7 (2019), 60868–60876. https://doi.org/10.1109/ACCESS.2019.2911641

Philip Rhodes. 2018. TIPP: parallel Delaunay triangulation for large-scale datasets. SSDBM '18: Proceedings of the 30th International Conference on Scientific and Statistical Database Management, 1–12. https://doi.org/10.1145/3221269.3223034

Guodong Rong, Tiow-Seng Tan, Thanh-Tung Cao, and Stephanus. 2008. Computing two-dimensional Delaunay triangulation using graphics hardware (I3D '08). Association for Computing Machinery, New York, NY, USA, 89–97. https://doi.org/10.1145/1342250.1342264

Johnathan Richard Shewchuk. 1996a. Robust adaptive floating-point geometric predicates. In Proceedings of the Twelfth Annual Symposium on Computational Geometry (Philadelphia, Pennsylvania, USA) (SCG '96). Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/237218.237337

Jonathan Richard Shewchuk. 1996b. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Applied Computational Geometry: Towards Geometric Engineering (WACG 1996) (Lecture Notes in Computer Science, Vol. 1148), Ming C. Lin and Dinesh Manocha (Eds.). Springer, 203–222. https://doi.org/10.1007/BFb0014497

Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Discrete & Computational Geometry 18, 3 (1997), 305–368. https://doi.org/10.1007/PL00009321

Jonathan Richard Shewchuk and Brielin C. Brown. 2015. Fast segment insertion and incremental construction of constrained Delaunay triangulations. Computational Geometry 48, 8 (2015), 554–574. https://doi.org/10.1016/j.comgeo.2015.04.006

Scott W Sloan. 1993. A fast algorithm for generating constrained Delaunay triangulations. Computers & Structures 47, 3 (1993), 441–450.

Hubert Wagener. 1988. Triangulating a monotone polygon in parallel. In Computational Geometry and its Applications, Hartmut Noltemeier (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–147.

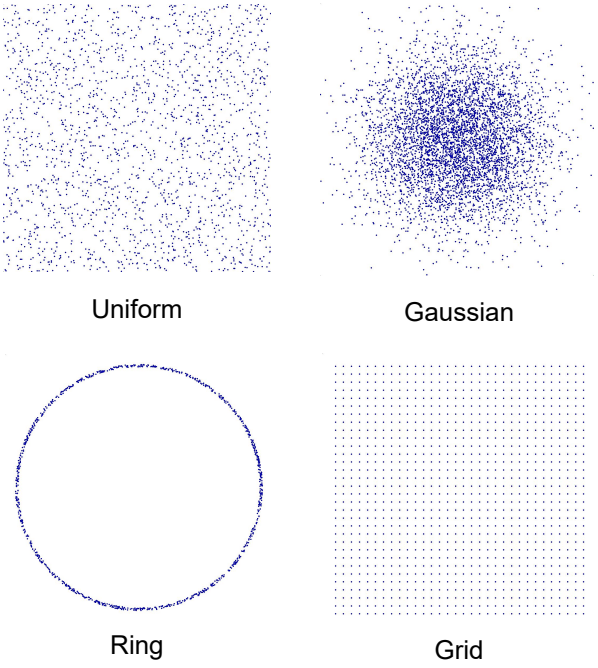


Fig. 9. **Synthetic.** Four benchmarks with 1M–9M vertices under four point distributions and no constraints. In *Ring*, vertices lie in a circular annulus with inner radius 0.45 and width 0.01. In *Grid*, vertices lie exactly on grid points.

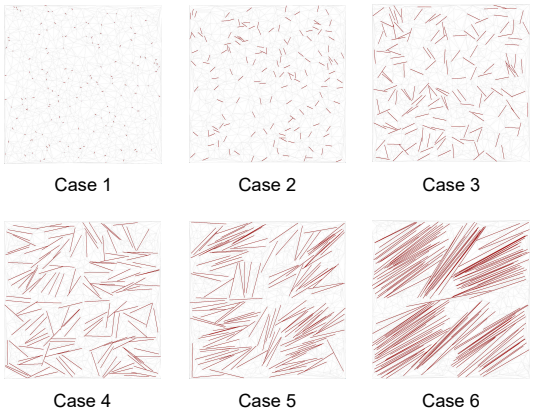


Fig. 10. **Synthetic-cons.** Six benchmarks with 1M uniformly distributed vertices and 150K constraints; cases vary the constraint-length distribution from short segments to scene-spanning constraints. For visualization only, we render 1,000 vertices and 150 constraints; the actual benchmarks are 1000× larger.

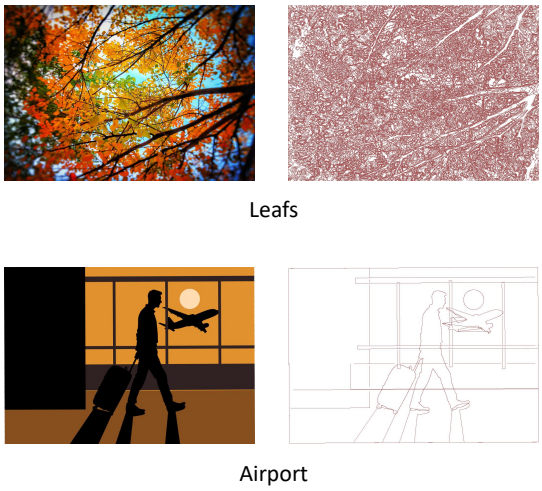


Fig. 11. **Leafs and Airport.** *Leafs* (top) is a vectorized image with many short constraints and an approximately uniform spatial distribution. *Airport* (bottom) is a vectorized image with a small number of long constraints and a non-uniform spatial distribution. In each row, the left shows the original image and the right shows the extracted constraints. The size and distribution of constraints in these two benchmarks are completely different.

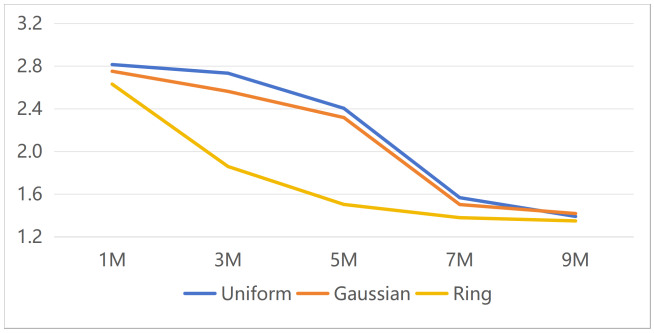


Fig. 12. **Speedups relative to** *gDel2D*. The vertical axis shows the speedup of *gCDT* over *gDel2D*. In Delaunay-only scenarios, *gCDT* achieves more than a 1.35X improvement.

Table 1. **Uniform.** All data represents runtime in milliseconds.

	gCDT	*gDel2D*	*PCDT*	*CGAL*	Sp.
1M	15.26	42.98	79.36	495.03	2.82
3M	22.49	61.52	180.67	1433.14	2.74
5M	33.23	79.96	315.88	2432.50	2.41
7M	63.88	100.21	477.73	3462.96	1.57
9M	87.25	121.48	594.62	4494.27	1.39

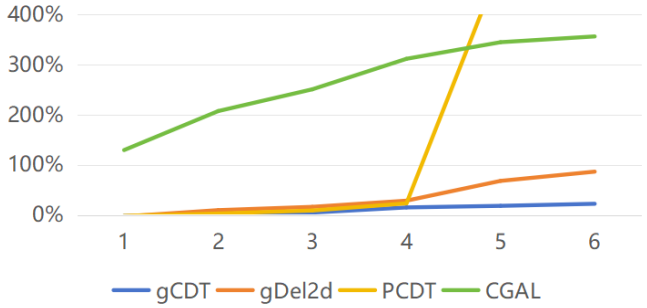


Fig. 13. **Impact of constraint length.** The vertical axis shows the relative performance degradation after constraint enforcement, where smaller values indicate lower sensitivity to constraints. The horizontal axis corresponds to the six **Synthetic-cons** benchmarks; larger indices indicate longer average constraint lengths. *gCDT* is substantially less affected by long constraints.

Table 2. **Gaussian and ring.** The performance of our algorithm is affected when the vertex distribution is poor.

	Gaussian			Ring		
	gCDT	*gDel2D*	Sp.	*gCDT*	*gDel2D*	Sp.
1M	15.36	42.30	2.75	16.66	43.87	2.63
3M	23.60	60.54	2.57	33.22	61.81	1.86
5M	34.46	79.93	2.32	53.56	80.61	1.51
7M	65.60	98.67	1.50	72.78	100.49	1.38
9M	85.65	121.60	1.42	93.22	125.94	1.35

Table 3. **Grid.** The left column indicates the edge length of the grid.

	gCDT	*gDel2D*	*PCDT*	*CGAL*	Sp.
256^2	10.54	71.36	73.04	44.91	4.26
512^2	15.98	160.44	79.55	180.21	4.98
768^2	17.43	174.18	92.20	402.70	5.29
1024^2	23.13	193.35	139.13	713.77	6.02

Table 4. **Synthetic-cons.** Our algorithm remains largely stable. PCDT must resort to double-precision calculations in Case 6.

	gCDT	*gDel2D*	*PCDT*	*CGAL*	Speedup
1	14.96	41.72	77.95	1136	2.79
2	15.53	47.13	81.67	1521	3.03
3	15.99	49.95	86.42	1736	3.12
4	17.56	55.24	97.78	2037	3.15
5	18.04	72.21	476.32	2201	4.00
6	18.67	80.14	1488.10*	2258	4.29

Table 5. **Real world data.** Our algorithm demonstrates multiple-fold performance improvements. In the Airport ($93K$) dataset, all GPU-based algorithms employ double-precision computation. PCDT fails to complete most of the tests.

	Size	*gCDT*	*gDel2D*	*PCDT*	*CGAL*	Sp.
Leafs	1.7M	112.01	213.4	349.1	3673.64	1.91
	3.4M	235.47	431.7	/	7602.22	1.83
Airport	47K	36.58	211.7	/	95.00	2.60
	93K	88.71	533.3	/	197.38	2.23
Letters	97K	129.06	1412.6	/	201.04	1.56
	970K	2.273s	56.07s	/	3.001s	1.32

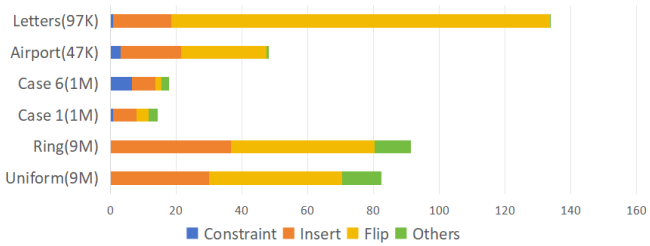


Fig. 14. **Runtime breakdown by phase.** After the constraint enforcement time is substantially reduced, edge flipping often becomes the primary bottleneck.

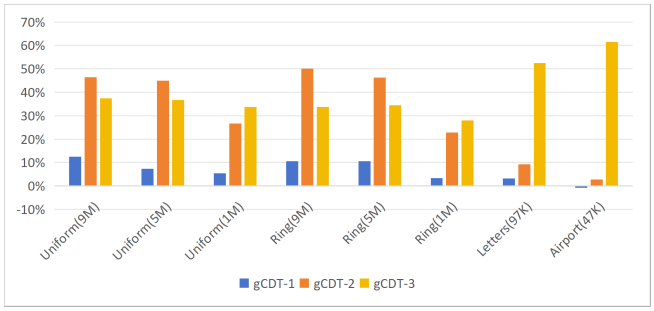


Fig. 15. **Performance degradation.** The vertical axis represents the percentage of performance degradation caused by the absence of a specific optimization. Higher values indicate that an optimization is more effective on the benchmark.

A PROOFS

A.1 Correctness of Distance-Ordered Polygon Retriangulation

Correctness follows the standard argument for distance-ordered triangulation of (weakly) monotone polygons [Wagener 1988]. The main requirement is to show that the produced diagonals do not cross. A key geometric property is a visibility relation to the base edge (v_0, v_n): for any $0 < i < j < n$, there exist points $\hat{v}_i, \hat{v}_j$ on (v_0, v_n) such that segments $(v_i, \hat{v}_i)$ and $(v_j, \hat{v}_j)$ lie inside the polygon and do not intersect (Figure 5(b)). This can be expressed via orientation predicates:

$$\begin{aligned} &Orient2D(v_0, v_i, v_j) > 0 \ \vee \ Orient2D(v_n, v_i, v_j) > 0, \\ &Orient2D(v_0, v_j, v_i) < 0 \ \vee \ Orient2D(v_n, v_j, v_i) < 0. \end{aligned} \tag{4}$$

Let v_i be a vertex minimizing $\tilde{d}_i$ over $0 < i < n$. Then adding diagonals (v_0, v_i) and (v_i, v_n) cannot intersect any future diagonal incident to vertices on either side: otherwise some vertex would lie strictly inside $\triangle(v_0, v_i, v_n)$, contradicting minimality of $\tilde{d}_i$. The two diagonals decompose the polygon into two independent subpolygons, and the argument applies recursively.

We omit implementation details identical to [Wagener 1988]. The auxiliary binary tree for computing (p_i, q_i) can be built efficiently in parallel using Karras' construction [Karras 2012], making the full triangulation GPU friendly.

A.2 Convexity of Holes

All holes produced by our procedure are convex polygons. The boundary consists only of constraint segments and edges generated by Section 6.4, and it cannot self-intersect. Moreover, every boundary vertex is visible only to its two incident boundary edges; if a vertex inside the hole existed, it would be visible to part of the boundary and would have been retained, contradicting the construction. If a hole were non-convex, it would contain a reflex boundary vertex whose incident edges span an angle $> 180°$. Such a configuration implies the existence of a mesh edge from that vertex entering the hole interior; the adjacent triangle would then intersect the boundary, making the vertex visible to a boundary portion, contradicting the visibility property. Hence no reflex angle exists and the hole must be convex.

B EXACT GEOMETRIC PREDICATES

The correctness and construction of a 2D Constrained Delaunay Triangulation fundamentally rely on two exact geometric predicates:

$Orient2D(A, B, C)$ determines the relative position of point C with respect to the directed line from A to B. It returns a positive, negative, or zero value if C lies to the left, right, or directly on the line, respectively.

$$Orient2D(A, B, C) = \begin{vmatrix} A_x - C_x & A_y - C_y \\ B_x - C_x & B_y - C_y \end{vmatrix} \tag{5}$$

$InCircle(A, B, C, D)$ determines the position of point D relative to the circumcircle of triangle $\triangle(A, B, C)$. It returns a positive, negative, or zero value if D lies inside, outside, or exactly on the circumcircle, respectively. The Delaunay condition for a triangulation can be stated as: for every interior edge, the $InCircle()$ test involving the two adjacent triangles and the opposite vertices must yield a non-positive result.

$$\begin{aligned} InCircle(A, B, C, P) &= \begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ P_x & P_y & P_x^2 + P_y^2 & 1 \end{vmatrix} \\ &= \begin{vmatrix} A_x - P_x & A_y - P_y & (A_x - P_x)^2 + (A_y - P_y)^2 \\ B_x - P_x & B_y - P_y & (B_x - P_x)^2 + (B_y - P_y)^2 \\ C_x - P_x & C_y - P_y & (C_x - P_x)^2 + (C_y - P_y)^2 \end{vmatrix} \end{aligned} \tag{6}$$