Meshless Voronoi on the GPU

Nicolas Ray, Dmitry Sokolov, Sylvain Lefebvre, Bruno Lévy

Meshless Voronoi on the GPU

NICOLAS RAY and DMITRY SOKOLOV, Université de Lorraine, CNRS, Inria, LORIA, France
SYLVAIN LEFEBVRE and BRUNO LÉVY, Université de Lorraine, CNRS, Inria, LORIA, France

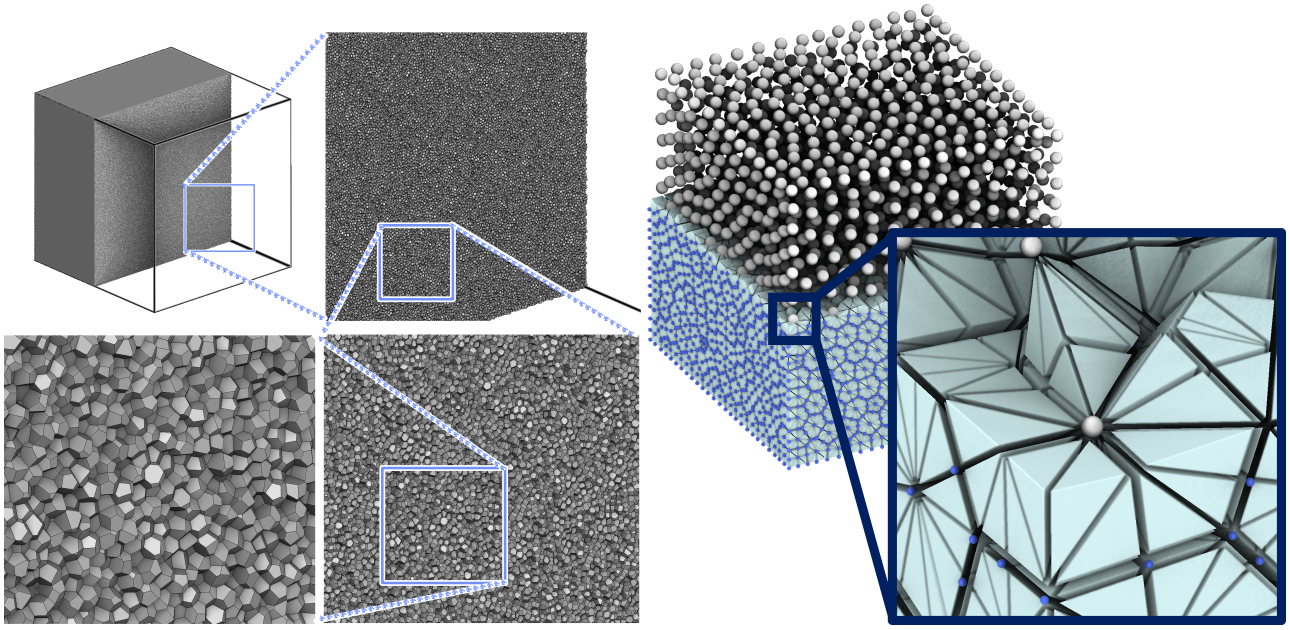

Fig. 1. Left: The 3*D* Voronoi diagram of 10 million points computed on the GPU in 800 ms (NVidia V100). We do not compute the tetrahedra, but in terms of equivalent computation speed, this corresponds to 84 million Delaunay tetrahedra per second. Right: We compute integrals over the Voronoi cells on the GPU, by decomposing them on-the-fly into tetrahedra, without using any combinatorial information.

We propose a GPU algorithm that computes a 3*D* Voronoi diagram. Our algorithm is tailored for applications that solely make use of the geometry of the Voronoi cells, such as Lloyd's relaxation used in meshing, or some numerical schemes used in fluid simulations and astrophysics. Since these applications only require the geometry of the Voronoi cells, they do not need the combinatorial mesh data structure computed by the classical algorithms (Bowyer-Watson). Thus, by exploiting the specific spatial distribution of the point-sets used in this type of applications, our algorithm computes each cell independently, in parallel, based on its nearest neighbors. In addition, we show how to compute integrals over the Voronoi cells by decomposing them on the fly into tetrahedra, without needing to compute any global combinatorial information. The advantages of our algorithm is that it is fast, very simple to implement, has constant memory usage per thread and does not need any synchronization primitive. These specificities make it particularly efficient on the GPU: it gains one order of magnitude as compared to the fastest state-of-the-art multi-core CPU implementations. To ease the reproducibility of our results, the full documented source code is included in the supplemental material.

CCS Concepts: • **Theory of computation → Computational geometry**; • **Computing methodologies → Parallel algorithms**;

Additional Key Words and Phrases: GPU, Voronoi diagrams

ACM Reference Format:
Nicolas Ray, Dmitry Sokolov, Sylvain Lefebvre, and Bruno Lévy. 2018. Meshless Voronoi on the GPU. *ACM Trans. Graph.* 37, 6, Article 265 (November 2018), 12 pages. https://doi.org/10.1145/3272127.3275092

1 INTRODUCTION

Voronoi diagrams and Delaunay triangulation are widely used because they exhibit interesting mathematical properties that are exploited both by the applications and the algorithms that generate them. There exists very efficient general-purpose implementations of the Delaunay triangulation, that can produce the tetrahedral mesh from any point-set (more on this below). However, producing a tetrahedral mesh requires some strategies to ensure that combinatorial decisions are globally coherent. In addition, parallel implementations need synchronization mechanisms to update the shared global combinatorial data structure.

We observe that for a certain class of applications, the global combinatorial information of the Voronoi diagram is actually not needed. Typical cases are Lloyd relaxation [Du et al. 1999; Liu et al.

Authors' addresses: Nicolas Ray, Nicolas.Ray@inria.fr; Dmitry Sokolov, Dmitry.Sokolov@univ-lorraine.fr, Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France; Sylvain Lefebvre, Sylvain.Lefebvre@inria.fr; Bruno Lévy, Bruno.Levy@inria.fr, Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France.

0730-0301/2018/11-ART265 $15.00
https://doi.org/10.1145/3272127.3275092

2009; Lloyd 1982], and some numerical schemes for fluid simulation [Brochu et al. 2010; Sin et al. 2009] (see also [Aanjaneya et al. 2017; de Goes et al. 2015; Gallouët and Mérigot 2017] that use power diagrams instead of Voronoi diagram). This also concerns Voronoi moving-mesh fluid simulators [Springel 2011], that are also applicable to some direct [White and Springel 1999] and inverse [Brenier et al. 2003] problems in cosmology.

We focus on this class of applications, that do not make use of the global combinatorics, and that use point-sets that are evenly distributed or that have a smoothly varying density:

- These applications solely use the Voronoi diagram as a partition of space, and compute integrals over cells (e.g Lloyd), or a weighted adjacency matrix (e.g FEM). In both cases, only the geometry of the Voronoi cells (and their direct neighbors for FEM) are required. It makes it possible to better separate the computation of each Voronoi cell, and compute the required information on-the-fly without needing to store any representation of the cell. In addition, it is no longer necessary to ensure that geometric predicates are globally coherent: while tetrahedra with zero volume may appear in the decomposition of the cell (Fig. 3), these will not contribute to the computation of the cell integral properties (e.g. center of mass);
- when the point-set is evenly distributed, or has a smoothly varying density, each Voronoi cell can be computed from its nearest neighbors: it will not share a facet with another cell that has its seed too far away (Fig. 4). Our assumption on the quality of the distribution is that only the k nearest neighbors will be sufficient to compute the Voronoi cell, with a relatively low value of k (typically 35 to 180).

In our context, the geometry of each Voronoi cell can be evaluated independently, using the local neighbors of the seed of the cell. This idea was initially proposed in [Rycroft 2009], it makes much easier to implement on highly parallel architectures such as GPUs. A regular or smoothly varying distribution of the point-set (blue noise, white noise) leads to similar running time/memory consumption for computing each cell, which makes the implementation simpler (constant memory usage), and more efficient on a SIMD architecture.

The key ingredients of our approach are:

- a method to compute each Voronoi cell independently, that gives parallelism for free;
- a minimalist data structure that replaces the combinatorial information and that can be easily updated and computed in each cell in parallel;
- a method to compute integrals over the Voronoi cells by decomposing them into tetrahedra, computed on the fly from the minimalist cell data structure.

The **main benefits** of our approach are its speed (an order of magnitude faster), and its simplicity ($\approx$ 500 lines of code).

The **main limitation** of our approach is that it is more specific than the Bowyer-Watson approach: it does not produce a global combinatorial data structure and it is only applicable to point-sets with even or smoothly varying distributions (white noise, blue noise).

Previous work

Several strategies for developing a parallel algorithm that computes a Delaunay triangulation (or Voronoi diagram) where proposed. Most of them are variants of the classical Bowyer-Watson algorithm [Bowyer 1981; Watson 1981] (point location and cavity retriangulation), where the point location phase is helped by spatial sorting [Amenta et al. 2003; Delage and Devillers 2018]. They can be sorted into three categories: coarse-grained parallelism, fine-grained parallelism and GPU implementations.

Coarse-grained parallelism. A first class of algorithms uses coarse-grained parallelism: they decompose the set of input points into subsets that are meshed in parallel, then merge the subsets. This strategy is well adapted to gigantic meshes computed on clusters [Alexander and Rainald 1995; De Cougny and Shephard 1999; González 2016; Nikos and Damian 2003; Rycroft 2009]. Besides the ability to work on clusters, another benefit of this method is that it can reuse an existing implementation to mesh each subset [Peterka et al. 2014].

Fine-grained parallelism. A second class of algorithms, still based on the Bowyer-Watson strategy, inserts points in parallel into the same mesh. Synchronization primitives are used to detect and resolve conflicting insertions. This strategy is well adapted to shared memory multi-cores. Using atomic memory operations supported by modern CPUs, it is possible in most case to avoid using costly OS synchronization primitives, replaced by light-weight user-space operations (spinlocks). The CGAL library [The CGAL Project 2018] uses a grid of spinlocks spatially organized [Batista et al. 2010]. It is also possible to attach the spinlocks to the combinatorial elements of the triangulation directly (as also suggested in [Batista et al. 2010]). The GEOGRAM library [Inria 2018] uses this strategy, with spinlocks attached to the tetrahedra. A completely different two-level approach is proposed in [Remacle 2017], and more recently, the same group of researchers proposed to insert batches of points in parallel in multiple partitionings computed from the Hilbert curve ordering of the input points [Marot et al. 2018]. The latter strategy scales up very well on machines with large number of cores (Intel MIC and AMD EPYC).

GPU implementations. The strategies mentioned above can be implemented on the GPU [Cao 2014; Cao et al. 2014], however, the Bowyer-Watson algorithm they are all based on is not well suited to GPU implementation. The main reason is the need to update a global combinatorial data structure, which results in both irregular memory accesses and diverging execution paths in the threads. It is however possible to compute restricted Voronoi diagrams by clipping each triangle of the surface [Fei et al. 2014]. For the sake of completeness, we also mention that discrete pixel-based) Voronoi diagrams can be computed on the GPU in 2D, using the Z-Buffer to determine to which cell each pixel belongs [Hoff et al. 1999], and related works devoted to compute distance fields both in 2D and 3D [Cuntz and Kolb 2007; Fischer and Gotsman 2006; Rong and Tan 2006; Schneider et al. 2010; Sigg et al. 2003; Sud et al. 2006, 2005, 2004]. A similar method is exploited to compute Centroidal Voronoi Tessellations [Fei et al. 2014; Rong et al. 2011]

The approach that we propose is based on a completely different strategy: it is based on the observation that given a $3D$ search data

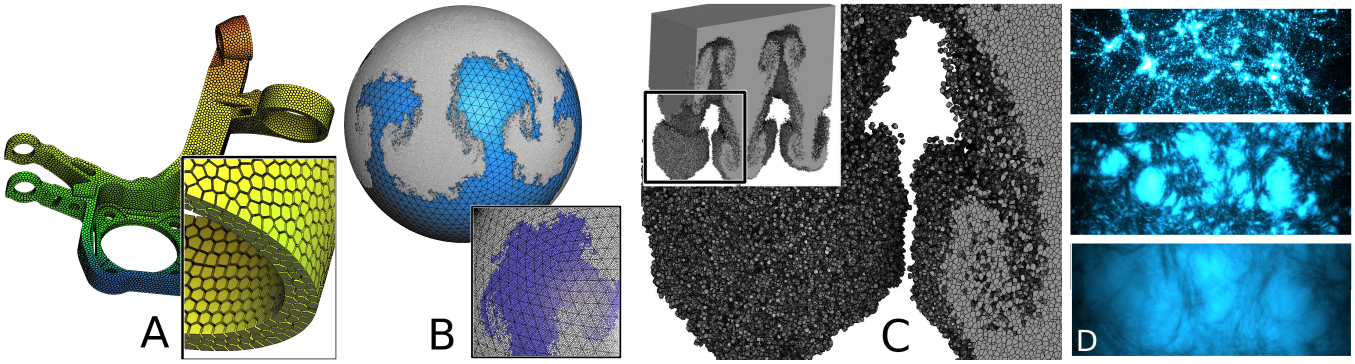

Fig. 2. Applications that use meshless Voronoi computation. Voronoi meshing and polyhedral finite elements (A) fluid simulation (B,C) and astrophysics (D). These applications solely need the geometry of the Voronoi cells, and do not need a global mesh topology.

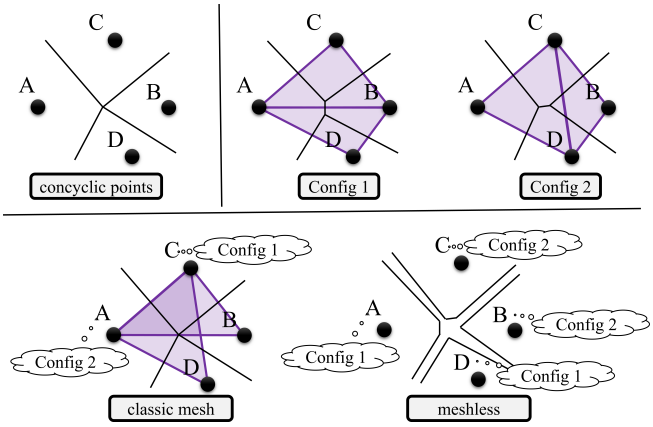

Fig. 3. Degenerate configurations illustrated in 2D: when four points are concyclic (upper left), there are two possible Delaunay triangulations (upper right). Classic Delaunay generation can produce incoherent mesh (lower-left) if geometric predicates are not consistent. Our meshless approach will just produce sometime a facet with zero area (or edge with zero-length in this 2D example).

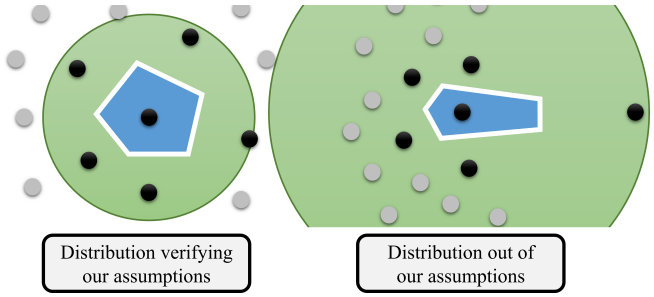

Fig. 4. Point-set distribution quality. To compute a Voronoi cell (blue), we need all the black points. We search them in the k nearest neighbors of the seed. For us, a good distribution will have a minimal number of points in the smallest disc (green) centered on the seed, and that contains all black points.

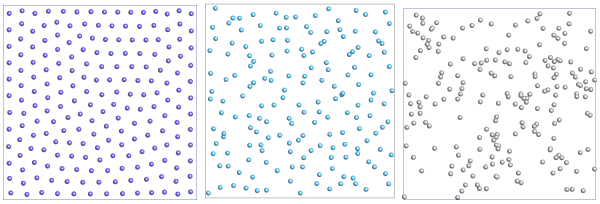

Fig. 5. The different classes of point distributions that we consider. From left to right: blue noise, perturbed grid and white noise.

structure for the K nearest neighbors problem, such as the one used in [Lévy and Bonneel 2013; Rycroft 2009], it is possible to compute all the Voronoi cells in parallel, without any need for synchronization. The implementation of VORO++ [Rycroft 2009] is very similar to our algorithm, but exploits the versatility of CPU to deal with its dynamic combinatorial data structure. We propose an alternative representation that is very GPU-friendly (see Section 5), thanks to the constant memory usage per thread and no need for any synchronization primitives.

The resulting algorithm is well adapted to the data-parallel model of the GPUs. However, it comes at the expense of being only applicable to cases where the number of Delaunay neighbors of each point remains smaller than a certain threshold (typically 35 to 190 in our implementation). Other cells need special treatment on the CPU.

Overview

The algorithm takes as input a point-set. It computes the Voronoi cells on-the-fly, and outputs integrals computed on them (e.g., barycenters, volumes, facet areas).

Our algorithm is based on a strategy similar to [Lévy and Bonneel 2013; Rycroft 2009], with special data structures well adapted to GPUs (more details and comparisons below). It is decomposed into the following two steps, detailed in the next two sections:

- compute the index of the k nearest points of each point §2
- compute Voronoi cells of each point from its nearest neighbors §3

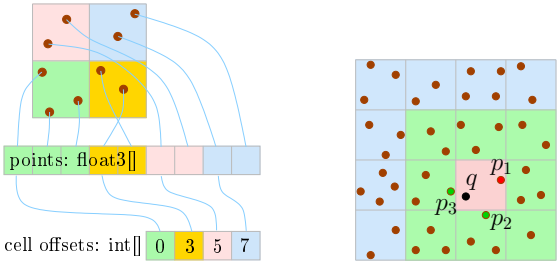

Fig. 6. Left image: in order to retrieve quickly all points inside a cell, we sort the point-set by corresponding cell id. Right image: to find k nearest neighbors, for a query point q we visit all neighboring cells in concentric rings.

We evaluate our algorithm with three types of point-set distributions, ranging from the best case (blue noise) to the worst case (white noise), shown in Figure 5: **Blue noise** — a white noise after 100 iterations of Lloyd's algorithm, **Perturbed grid** — points are initialized from a regular grid perturbed by a uniform law in the range of a grid cell, **White noise** — coordinates of each point is defined by a uniform law. In the Appendix, we present and discuss a certified version of the algorithm.

2 k-NEAREST NEIGHBORS

The first step of our algorithm consists in computing the k nearest neighbors of each point. Several general-purpose methods were proposed on the GPU. Most of them are based on the observation that a simple brute-force strategy can make use of the massive parallel horsepower of the GPU [Barrientos et al. 2017; Garcia et al. 2008, 2010; Li and Amenta 2015]. In our case, since our data is low-dimensional ($3D$), we prefer to use a grid-based strategy, as often done in fluid simulation [Hoetzlein 2014].

This step takes as input the set of n 3d points and produces a $k \cdot n$ array of indices (k neighbors for each of n points). Note that for $n = 10^7$ and $k = 40$ the resulting array takes 1.5Gb of memory (with 32-bit integers), and an obvious optimization would be to generate the array chunk by chunk by interleaving calls to Voronoi cells computation with k-NN search. However, for the sake of simplicity, to ease testing several alternatives for each step, we generate the complete array.

The pseudo-code for this step is detailed in Algorithm 1. First of all, we embed the point-set in a 3d grid (with ≈ 5 points per cell on average). Note that for each point we can obtain the corresponding cell id by a simple rounding of its coordinates. Then we sort the points by ascending cell id order, it allows us to quickly retrieve all points inside a given cell (left image of Fig. 6).

Then for each query point, we visit its neighbors in concentric cell rings. We maintain an array of k candidates, and if a distance to a neighbor is inferior to the maximum distance in the array, we update the maximum element. The array represents a binary max-heap structure that is the most efficient for this usage.

Right image of the Fig. 6 shows an illustration for the case when we want to find 2 nearest neighbors of the query point q. As we store k-NN candidates in a max-heap, here it will be a simple ordered pair. We start with an 2-element max-heap $(\varnothing, \varnothing)$. After visiting all points of the starting cell, the max-heap will be $(\varnothing, p_1)$. Then we visit first neighboring ring (shown in green). After visiting all the points in the ring, the max-heap will be (p_2, p_3). All following rings are not visited since the minimum distance from the point q to all other rings is superior to the distance to first element of the heap.

Algorithm 1: k-nearest neighbors computation

Input: Point-set (n float3 values)
Output: Reordered point-set and a k-NN array ($k \cdot n$ integers)

```
Define the grid resolution;
Sort the point-set by the cell id ;                    // see Fig. 6
int knn[n][k];
foreach q in the point-set do
    h ← [∅, ... ∅] ;    // k points infinitely far from q
    r ← 0;
    foreach point p in the r-ring of q do
        if ||p − q||² < ||h[0] − q||² then
            h[0] ← p ;      // replace the farthest element
            Max-Heapify(h, 0) ;          // repair the heap
        end
        if ||h[0] − q||² > min distance to the next ring then
            r + + ;
        else break ;
    end
    HeapSort(h);
    knn[q] ← h;
end
return point-set, knn;
```

3 COMPUTING THE DIAGRAM

The Voronoi cell of a seed i is by definition the subset of $\mathbb{R}^3$ made of the points that are closer to the seed i than to any other seed j of the point-set. From this definition, it is clear that the Voronoi cell corresponds to the intersection of a set of half-spaces (§3.1). Each of them is bounded by the medial plane of the segment i, j and contains the seed i. We stick to this definition and explicitly compute the intersection, as detailed in algorithm 2. Each cell is initialized by the bounding box of the point-set (§3.2) and is iteratively clipped by the half-spaces defined by its k nearest neighbors (§3.3). Once all the half-spaces are processed or the security radius is reached, we check the validity of the computations (§3.4) and, in case of success, we perform the computations on the cell required by the application (§3.5).

3.1 Voronoi cells as half-space intersections and radius of security

Suppose that you want to compute the Voronoi cell $Vor(\mathbf{x}_i)$ of the point $\mathbf{x}_i$ in the Voronoi diagram $Vor(\mathbf{x}_1, \ldots \mathbf{x}_n)$. The Voronoi cell is

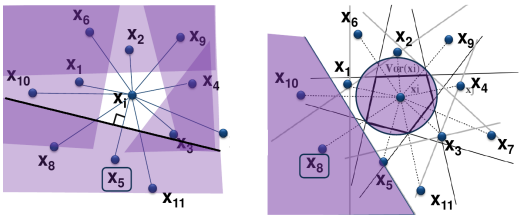

Fig. 7. Left: a Voronoi cell corresponds to the intersection of half-spaces; Right: the "Radius of Security" criterion selects the points that contribute to a Voronoi cell.

defined by:

$$\begin{aligned} \mathbf{x} \in Vor(\mathbf{x}_i) \quad &\Leftrightarrow \quad d(\mathbf{x},\mathbf{x}_i) < d(\mathbf{x},\mathbf{x}_j) \qquad \forall j \\ &\Leftrightarrow \quad \mathbf{x} \in \cap_j \{\mathbf{x} | d(\mathbf{x},\mathbf{x}_i) < d(\mathbf{x},\mathbf{x}_j)\} \\ &\Leftrightarrow \quad \mathbf{x} \in \cap_j \Pi^+(i,j), \end{aligned}$$

where $\Pi^+(i,j) = \{\mathbf{x} | d(\mathbf{x},\mathbf{x}_i) < d(\mathbf{x},\mathbf{x}_j)\}$ denotes the set of points nearer to $\mathbf{x}_i$ than $\mathbf{x}_j$. Note that $\Pi^+(i,j)$ is a half-space, bounded by the bisector of points $\mathbf{x}_i$ and $\mathbf{x}_j$. In Figure 7-left, the bisector of $\mathbf{x}_i$ and $\mathbf{x}_5$ is highlighted. Each bisector clips a half-space (grayed out in the Figure). Clearly, using this definition directly gives no practical algorithm, since each Voronoi cell needs to be clipped by the bisectors defined by all the other points, leading to at least an $O(n^2)$ algorithm. However, a simple geometric criterion called the "radius of security" [Lévy and Bonneel 2013], also used in VORO++ [Rycroft 2009], makes it possible to select a small subset of the points that truly contribute to the Voronoi cell:

Still supposing that we compute the Voronoi cell of point $\mathbf{x}_i$, we suppose in addition that the other points $\mathbf{x}_1, \ldots \mathbf{x}_n$ are ordered in increasing distances from $\mathbf{x}_i$ (which is the case with the k-NN algorithm presented in the previous section). At each clipping operation, we consider a bounding ball of the cell computed so-far, centered on $\mathbf{x}_i$ (see Figure 7-right). Clearly, if the current point (say $\mathbf{x}_8$ as in the figure) is further away from $\mathbf{x}_i$ than twice the radius of the bounding ball, then its clipping plane cannot touch the ball, thus it will not change the Voronoi cell (can be easily proven using the triangular inequality, see [Lévy and Bonneel 2013]). The other points are even further away (remember they come in increasing distance order), thus this means that as soon as this criterion is met, what we have computed exactly corresponds to the Voronoi cell.

3.2 Data structure and its initialization

The data structure that represents a cell (convex polyhedron) is in general non-trivial, since a cell can have faces with arbitrary number of edges. However, the dual of a cell can be represented by a simple triangle mesh. In other words, to represent the dual, a couple of arrays will suffice. Thus we represent the connectivity of the cell in dual form. Note that we do not assign any geometry to the dual, it represents only the connectivity.

For each Voronoi cell we use the following notations:

- $\mathcal{P} = \{P_i\}$: the set of half-space equations such that (x, y, z) is in the half-space $\mathcal{P}_i$ iff $a_i.x + b_i.y + c_i.z + d_i > 0$ where (a_i, b_i, c_i, d_i) represent the coefficients of the equation of the half-space $\mathcal{P}_i$;

Algorithm 2: Computation of integrals on a Voronoi diagram

Input: float3 points[n], int knn[n][k]
Output: Depends on the application, can be (per cell) barycenter, pressure, forces, volume…

```
bb ← BoundingBox(points);
for s ← 0 to n do
    cc ← ConvexCell(bb) ;                          // §3.2
    for p ← 0 to k do
        cc ← clipByPlane (cc, knn[s][p]) ;         // §3.3
        if securityRadiusReached(cc) then          // §3.1
            break;
    end
    if isValid (cc) then                           // §3.4
        computeOutput(cc);                         // §3.5
    else
        throw an error;
end
```

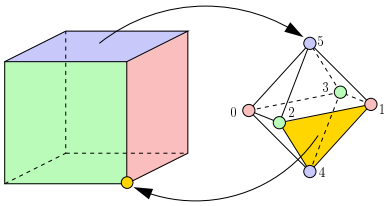

Fig. 8. Convex cells are initialized by the bounding box (left). The connectivity of the cell is stored by its dual triangle mesh (right). Note that we do not assign any geometry to the dual.

- $\mathcal{T} = \{T_i\}$: the connectivity of the cell stored in dual form. Each (dual) triangle is represented its (dual) vertices i.e a triplet of clipping plane IDs (primal facets) that intersects at the location of the corresponding (primal) vertex.

We initialize the Voronoi cell by the bounding box of the simulation domain. To do so, we create the clipping plane equations and the connectivity array as follows (refer to Fig. 8):

$$\begin{aligned} \mathcal{P} \leftarrow \{&(1,0,0,-x_{\min}),(-1,0,0,x_{\max}),\\ &(0,1,0,-y_{\min}),(0,-1,0,y_{\max}),\\ &(0,0,1,-z_{\min}),(0,0,-1,z_{\max})\},\\ \mathcal{T} \leftarrow \{&(2,5,0),(5,3,0),(1,5,2),(5,1,3),\\ &(4,2,0),(4,0,3),(2,4,1),(4,3,1)\}, \end{aligned}$$

where $x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}, z_{\max}$ are the minimum and maximum coordinates of the bounding box.

Note that we do not store any vertex coordinate explicitly: the i-th vertex can be obtained as the intersection between the three planes whose indices are given by the triplet T_i. The next step is to iteratively clip the cell by the half-spaces defined by the medians between the seed point and its neighbors.

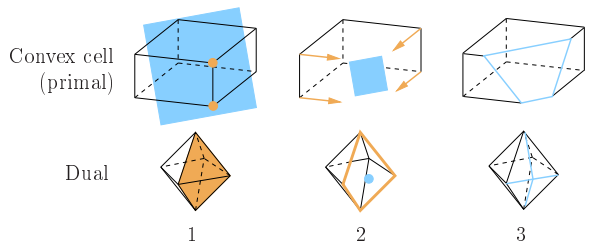

Fig. 9. Clipping a cell, illustrated both on primal (upper row) and dual (lower row). First we identify and remove the vertices (shown in orange) located on the "wrong" side of the green clipping plane (column 1). Then we create an (empty) new facet (column 2) and we connect the new facet to the mesh by introducing new vertices (column 3).

Algorithm 3: Clip a convex cell by a plane

```
1  Function clipByPlane:
       Input: convex cell (T, P), plane equation p (float4)
       Output: clipped convex cell
2      R ← ∅;
3      foreach (u, v, w) ∈ T do
           /* for all vertices of the cell                    */
4          (x, y, z) ← intersectPlanes(P_u, P_v, P_w);
5          if (x, y, z, 1) · p > 0 then          // is it clipped?
6              T ← T \ {(u, v, w)};          // remove it from T
7              R ← R ⋃ {(u, v, w)};              // add it to R
8          end
9      end
10     if R ≠ ∅ then                        // is anything clipped?
11         P.push(p);                      // add new plane equation
12         ∂R ← computeBoundary (R);
13         foreach edge (s, t) ∈ ∂R do
14             T ← T ⋃ {(s, t, p)};        // insert new triangle
15         end
16     end
17     return T, P;
```

3.3 Clipping

To clip the convex cell by a new half-space, in terms of the primal we do the following (refer to the upper row of the Fig 9):

(1) We identify the subset $\mathcal{R} \subset \mathcal{T}$ of vertices that belong to the "wrong" side of the half-space and we remove $\mathcal{R}$ from $\mathcal{T}$;
(2) we create a new empty facet;
(3) for each dangling edge (having exactly one of its extremities in $\mathcal{R}$), we add a new vertex to the new facet.

Since the connectivity is represented in dual form, we need now to rewrite the procedure in terms of the dual (bottom row of the Fig 9 and pseudo-code in Algorithm 3):

(1) We identify triangles $\mathcal{R}$ and remove them from the mesh, it leaves a "hole" in the mesh (lines 3–9 of Alg. 3);
(2) we create a new vertex (recall that it corresponds to the new cell face) (line 11 of Alg. 3);

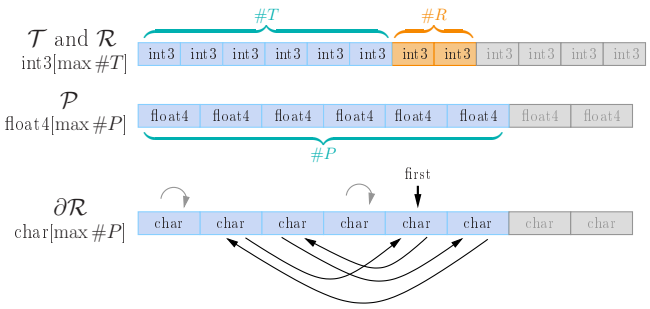

Fig. 10. The sets $\mathcal{T}$ and $\mathcal{R}$ are stored in a shared memory array of constant size max $\#T$, the variables $\#T$ and $\#R$ are the corresponding element counters. $\mathcal{P}$ and $\partial\mathcal{R}$ are also stored in shared memory arrays of constant size max $\#P$, and $\#P$ is the number of stored plane equations.

Algorithm 4: Compute the boundary of a triangulated topological disc

```
1  Function computeBoundary:
       Input: triangulated topological disc R
       Output: boundary ∂R (circular list of vertices)
2      ∂R ← ∅;
3      while R ≠ ∅ do
4          foreach r ∈ R do
5              if isSimpleCycle (r + ∂R) then
6                  ∂R ← ∂R + r;
7                  R ← R \ {r};
8                  break;
9              end
10         end
11     end
12     return ∂R;
```

(3) for each boundary edge of $\mathcal{R}$ we create a triangle incident to the vertex (lines 13–15 of Alg. 3).

Steps 1 and 2 are straightforward to implement. We store the sets $\mathcal{T}$ and $\mathcal{R}$ in a constant-size shared array, two variables $\#T$ and $\#R$ provide the corresponding element counters (Fig. 10). If the i-th triangle is to be removed from $\mathcal{T}$ and moved to $\mathcal{R}$, we decrement $\#T$, increment $\#R$ and we swap $T[i] \leftrightarrow T[\#T]$.

For step 3, we need to find the boundary edges of $\mathcal{R}$ (line 12 of Alg. 3). It is easier to consider the triangulated surface defined by the dual of $\mathcal{R}$. The boundary of this triangulated surface is the sum of all the elementary cycles associated to each triangle. We compute it by merging them with the strategy described in Alg. 4 and illustrated in Fig. 11: we initialize an empty cycle and iteratively add triangles as long as the cycle is kept simple. We want the cycle to be simple because it can be represented by a simple circular list. This list is stored in a constant size shared memory array (bottom row of Fig. 10), that makes it possible to add triangles in constant time: the triangle vertex IDs point directly to the right location in the circular list.

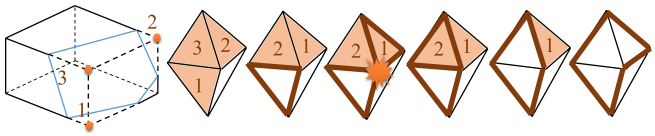

Fig. 11. Finding the boundary of $\mathcal{R}$: We initialize with the boundary of 1, and remove it form $\mathcal{R}$. We try to add the boundary of the new 1, but it would fail to produce a simple cycle. We try to add the boundary of 2 and it works, so we remove 2 from $\mathcal{R}$. Now we finish by adding the boundary of 1 and emptying $\mathcal{R}$.

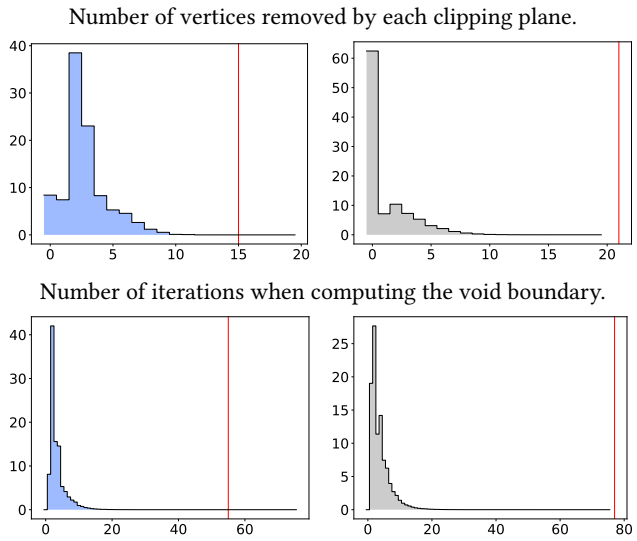

Fig. 12. Statistics of the hole filling algorithm with blue noise (left) and white noise (right). Red lines show the largest encountered value. Red lines show the largest encountered value.

It is reasonable to assume that we can reconstruct the boundary with simple cycles because $\mathcal{R}$ is defined as the intersection of a convex cell with an half-space, that basically decomposes the edge graph into exactly two connected components $\mathcal{R}$ and $\mathcal{T}-\mathcal{R}$. Failures during this step will only happen if two conditions are encountered simultaneously: (1) there are some Voronoi vertices located on the clipping plane or affected to the wrong side due to numerical estimation of predicates (2) there exists other Voronoi vertices that have a conflict with them. With $10M$ points, it typically occurs ten times with white noise, two times with a perturbed grid, and never with blue noise.

The overall complexity for a given cell is in $O(n^2)$ where n denotes the number of dual triangles on the boundary of the cell. Note that in our case, we have a very small number of triangles in $\mathcal{R}$. Fig. 12 shows histograms on the number of iterations for white and blue noise.

3.4 Exit status

For each Voronoi cell, our algorithm terminates with one of the following outcomes:

- **Success:** all clippings are done, the Voronoi cell is successfully constructed;
- **Security radius was not reached:** All neighbors are visited, but the security radius was not reached. We are not sure that the cell is the Voronoi cell.
- **Clip planes overflow:** half-spaces do not fit into the array $\mathcal{P}$.
- **Vertices overflow:** the cell has more vertices than we can store in the array $\mathcal{T}$;
- **Boundary error:** the computeBoundary procedure detected inconsistent data (infinite loop in line 3 of Alg. 4). This behavior is due to the numerical evaluation of predicates that do not ensure that $\mathcal{R}$ is a triangulated topological disc.

All failure cases can be fixed, but not for free. The first three types of failure can be fixed by increasing k, max $\#R$ and max $\#T$, but it impacts the performances §5.1. Fixing boundary errors requires to recompute the cell with robust predicates. In our experimental results, this concerns in the worst case 10 cells out of 10 millions. They can be fixed in a post-processing phase on the CPU. There are two main options to do that, either use the very same algorithm implemented on the CPU, but with a symbolically-perturbed exact predicate. Since the used predicate corresponds to the classical `in_sphere` (more on this in Appendix A), it is readily available (in GEOGRAM, CGAL, Shewchuk's predicates). A simpler-to-program alternative, used in the supplemental material of this article, is based on the simple observation that the Voronoi cell of $\mathbf{x}_i$ can be also obtained by computing a Delaunay triangulation of the point $\mathbf{x}_i$ and its neighbors, and taking the dual cell of vertex i. Clearly, it is equivalent to the first option, but maybe simpler in terms of software engineering since Delaunay triangulation codes are readily available (GEOGRAM, CGAL, tetgen …).

3.5 Computing barycenters and volumes

Once the Voronoi cell is computed, we need to evaluate its barycenter and volume. To do so, we decompose the cell into tetrahedra, evaluate the barycenter and volume of each tetrahedron, and derive the cell barycenter and volume of the cell.

A natural decomposition would triangulate Voronoi facets and produce a tetrahedron for each triangle by adding the seed of the cell. Unfortunately, iterating on Voronoi facets from our minimalist data structure is not free, so we seek for a method that iterates on Voronoi vertices and triangulates the dual. The difficulty is now to define a geometry for the dual.

We define the geometry of the dual in such a way that it can be computed directly from the seed of the cell and the three planes that intersect on each Voronoi vertex. To produce six tetrahedra for each vertex of the cell, we orthogonally project the Voronoi seed on each plane, and on each intersection of two planes (Fig. 13–left), then we create tetrahedra (Fig. 13–middle), to have a decomposition of the volume associated to the Voronoi vertex (Fig. 13–right).

Using the orthogonal projection of the seed onto the support (plane or line) of facets and edges of the cell ensures that tetrahedra produced from different Voronoi vertices will match (Fig. 14–left). However, there is no guarantee that these points will be located inside the facet or the edge: they can be located somewhere else on

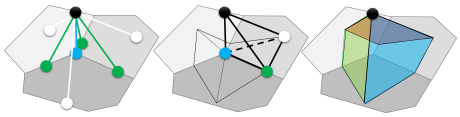

Fig. 13. Geometry of the tet decomposition associated to a Voronoi vertex (blue) defined as the intersection of the three (gray) planes. Left: the Voronoi seed (black) is projected on adjacent planes (white) and their intersections (green). Middle: each tetrahedron is produced by combining one vertex of each type. Right: volume associated to the Voronoi vertex.

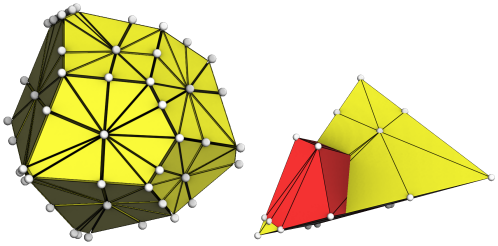

Fig. 14. Voronoi cells are decomposed into a set of tetrahedra (left). Our decomposition may be larger than the Voronoi cell, but is suitable for computing integrals because all extra volume is canceled by tetrahedra with negative signed volume.

their support plane or line (Fig. 14–right). This would be an issue if the objective was to export the geometry, but it does not matter for computing integrals because all extra volume of our decomposition is balanced by tetrahedra with negative signed volume, just like the classical computation that uses the Stokes formula for computing the volume inside a closed triangulated surface.

4 OPTIMIZATIONS

The most important parameter of our algorithm is the maximum number of neighbors required to clip the Voronoi cells. It impacts directly the construction of the k nearest neighbors, the GPU memory usage, and the clipping of the Voronoi cells. The size of arrays $\#\mathcal{P}$ and $\#\mathcal{T}$ are less critical, but should fit in some fast memory.

To determine these parameters, we test a very favorable case (blue noise) and the worst case that we are able to work with (white noise). We obtain the distributions of Fig.15 by running our algorithm on a point-set of $1M$ points. Using minimal parameters that produce all Voronoi cells with white noise is safe, but it is better to optimize at least the number of neighbors according to the application, as demonstrated in the results section.

GPU specific

The ability of our algorithm to compute each Voronoi cell independently is a good starting point for a GPU implementation. Moreover, the fixed per-thread memory usage (Fig. 10) is very GPU friendly. The only GPU specific optimization was to store all the data of Fig. 10 in the fast shared memory of the GPU, and access global memory only to get the neighbors, one by one. The number of thread by blocks was set to 16 in the reported experimental results.

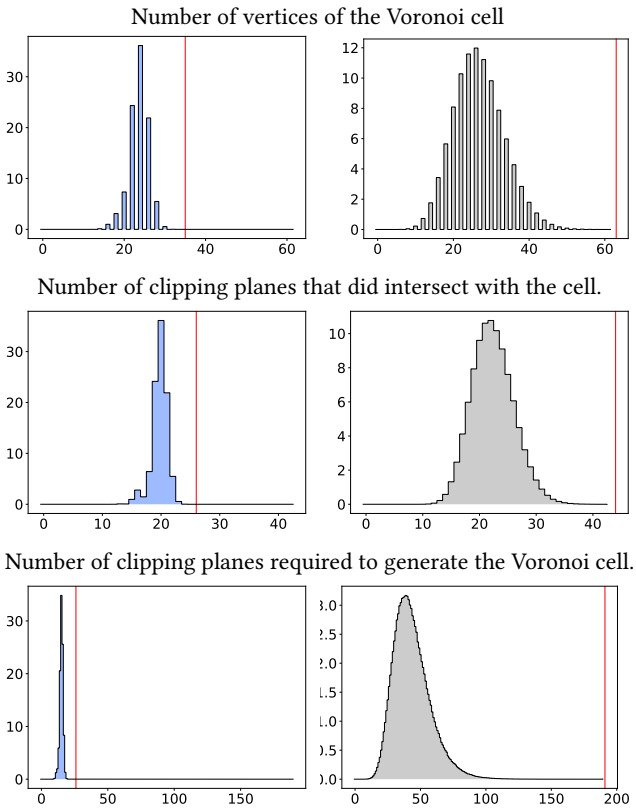

Fig. 15. Statistics to of Voronoi cell with blue noise (left) and white noise (right). Red lines show the largest encountered value.

5 RESULTS

We first evaluate the performance of our algorithm with different point-set size and type of distribution, and compare them with the state of the art of general purpose Voronoi diagram algorithms. We also observe how it behaves and can be optimized in the context of Lloyd relaxation as an example. We then discuss how it can be used and/or adapted to other applications.

5.1 Performance

The time consuming steps of our algorithm are the construction of the k nearest neighbors and the clipping of the Voronoi cells. We report timings for the three point-set distributions (blue, perturbed grid and white), for different number of points (100K, 1M and 10M), on two graphics hardware (GTX1080 and a V100). In table 1, the array sizes and number of neighbors are chosen to reach the security radius on all cells, and in all configurations. In table 2, we optimize these parameters according to the type of point-set distribution. We did not obtain timings for the GTX1080 with 10M points and the settings for white noise because computing all k nearest neighbors before starting to produce Voronoi cells kernel requires too much memory.

It is simple to observe that the algorithm scales linearly with the number of points, and that the V100 is approximately three times

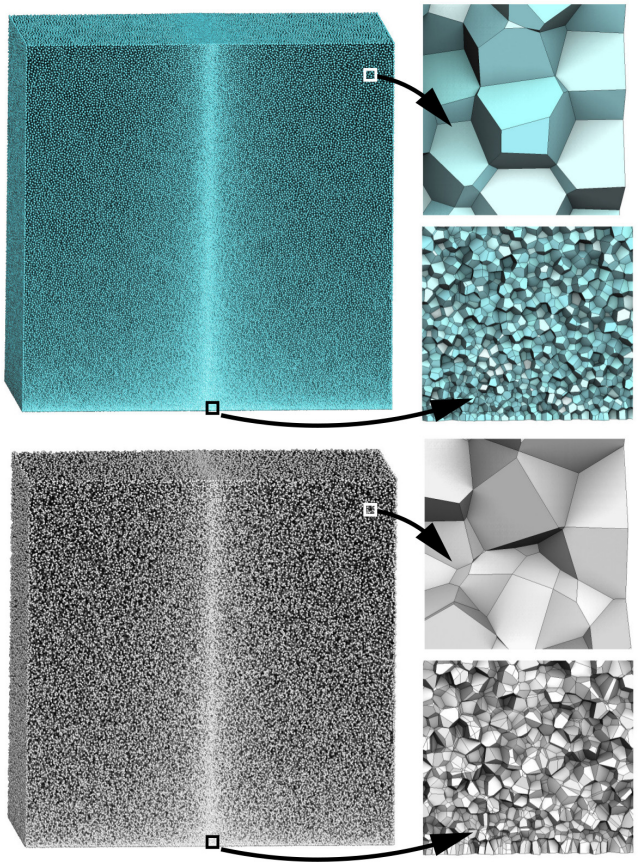

Fig. 16. A point-set of 10 million points with varying density. Closeup on a cross-section of the Voronoi diagram. Top: blue noise. Bottom: white noise.

faster than the GTX1080 on all experiments. We also observe that it is very important to tune the algorithm parameters to exploit the quality of the noise. Without such optimizations, blue noise is at best 30% faster than white noise (V100, 10M points). With better parameters, we achieve a speed-up of 1000% (green line of table 2).

The most important parameter to tune is k because the knn construction has to visit more cells, needs to sorts more points, and accesses more the global slow memory. The clipping part also has to access more the slow memory and has to test where the Voronoi vertices are with respect to each new clipping plane. Other parameters max $\#P$ and max $\#T$ do not impact the timings much, as long as the arrays fits into the fast shared memory of the GPU. In our implementation, we use 32 threads per block, to have enough shared memory to store our minimalist data structure.

When the density of points varies in space (see Fig. 16), we measured slightly slower timings (still on the V100): for white noise, knn: 4.496 s, clipping: 3.407 s and for the blue noise, knn: 2.98 s, clipping: 0.519 s. For a smoothly varying density of points, the number of neighbors required to reach the security radius remains small, and our algorithm continues to behave efficiently.

Table 1. Timings for different types of data-sets with conservative settings ($k = 190$, max $\#T = 96$, max $\#P = 64$). The timings are given in seconds.

	100K points		1M points		10M points	
	knn	clip	knn	clip	knn	clip
Tesla V100:						
white noise	0.041	0.037	0.360	0.333	3.387	3.334
perturbed grid	0.038	0.023	0.338	0.209	3.088	2.124
blue noise	0.033	0.007	0.308	0.079	2.825	0.688
GTX1080:						
white noise	0.134	0.089	1.275	0.919	n/a	n/a
perturbed grid	0.141	0.060	1.175	0.579	n/a	n/a
blue noise	0.122	0.022	1.033	0.192	n/a	n/a

Table 2. Timings for different types of data-sets with tuned settings (seconds). white noise ($k = 190$, max $\#T = 96$, max $\#P = 64$) perturbed grid ($k = 90$, max $\#T = 96$, max $\#P = 50$) blue noise($k = 35$, max $\#T = 96$, max $\#P = 32$)

	100K points		1M points		10M points	
	knn	clip	knn	clip	knn	clip
Tesla V100:						
white noise	0.041	0.037	0.360	0.333	3.387	3.334
perturbed grid	0.010	0.018	0.091	0.172	0.772	1.699
blue noise	0.002	0.004	0.016	0.046	0.158	0.417
GTX1080:						
white noise	0.134	0.089	1.275	0.919	n/a	n/a
perturbed grid	0.037	0.051	0.301	0.458	2.575	4.708
blue noise	0.005	0.011	0.052	0.117	0.520	1.135

Table 3. GEOGRAM vs CGAL vs VORO++, 12 cores Intel i7-6800K CPU @ 3.40GHz. Note that VORO++ uses a single core only.

	100K points	1M points	10M points
GEOGRAM 1.6.5:			
white noise	0.2	1.6	15.5
perturbed grid	0.2	1.6	15.5
blue noise	0.2	1.6	15.5
CGAL 4.11:			
white noise	0.15	1.3	12.5
perturbed grid	0.15	1.3	12.5
blue noise	0.15	1.3	12.6
VORO++ 0.4.6:			
white noise	1.6	17.2	173.8
perturbed grid	1.2	12.7	127.4
blue noise	0.9	10.0	98.8

The algorithm implemented in VORO++ [Rycroft 2009] is very similar to ours, and competes with other CPU methods. With minimal optimizations (see code in Supplemental material), running our algorithm on the CPU gives similar results to VORO++. We also compare our performance with the general-purpose parallel implementations of Voronoi diagram available in CGAL 4.11 and

GEOGRAM 6.5 (Table 3)[1]. We observe that the type of point-set distribution does not impact their performance. In our worst case (white noise), our implementation runs 30% faster than CGAL, but with tuned parameters and blue noise, it runs 16 times faster. Compared with the results reported in [Marot et al. 2018] on many-core EPYC and MIC processors, we are up to 2x faster[2]. Note however that their method, based on a Bowyer-Watson kernel, computes the global combinatorial information. However, their spatial sorting is probably like our approach sensitive to very irregular point-set distributions.

5.2 Example: Lloyd's algorithm

The Lloyd algorithm is a typical example where computing the Voronoi cells by our algorithm can greatly impact performance. For this application, we only optimize k, the number of neighbors.

We started the simulation with a white noise and observe (Fig. 17) how the size of the neighborhood is impacted by the type of noise. The first 5 iterations require quite a large neighborhood, so we run them with the white noise settings. After those iterations, the noise becomes uniform enough to be run much faster with the blue noise settings.

We also observe that the upper bound computed by the security radius is quite close to the minimal size of the neighborhood for blue noise (35 instead of 30) and for white noise (200 instead of 150).

6 DISCUSSION AND CONCLUSION

The algorithm is fast and can be directly exploited by some applications such as Voronoi moving-mesh fluids dynamics [Brochu et al. 2010; Springel 2011]. However, we believe that the real potential of the approach is that the algorithm is simple enough to be easily adapted to strongly related problems that are used in many applications developed during the last decade. For example, some recent fluids simulations are working with Laguerre diagram instead of Voronoi diagrams [Aanjaneya et al. 2017; de Goes et al. 2015; Gallouët and Mérigot 2017], that can also be constructed by iteratively clipping a convex cell. Another possible extension would be to produce Voronoi cells restricted to a tetrahedral volume: each Voronoi cell have to be clipped by each tetrahedra, and can modulate the integrals by attributes associated to the tetrahedra such as a frame field in LpCVT [Lévy and Liu 2010].

For applications like surface reconstruction or estimation of the medial axis, our algorithm will not be able to compute the Voronoi diagram from this type of points distribution... however, it is not always clear that the information exploited by the algorithm requires the Voronoi cell to be completely computed. For example, the ϵ-sampling criterion for surface reconstruction [Amenta et al. 2002] only requires the closest neighbors of the cell [Boltcheva and Lévy 2017].

Since it generates all the Voronoi vertices, our algorithm could also be used to compute the Delaunay triangulation: for each Voronoi vertex, the index of the current seed and the three indices of the bisectors on which the Voronoi vertex is located correspond to a Delaunay tetrahedron. Then each Delaunay tetrahedron is seen 4 times (one can pick the one for which the current seed has the lowest index among the 4). To generate the tetrahedra, robust and globally coherent geometric predicates will be required (see Figure 3). The robust version of the algorithm is detailed in the Appendix.

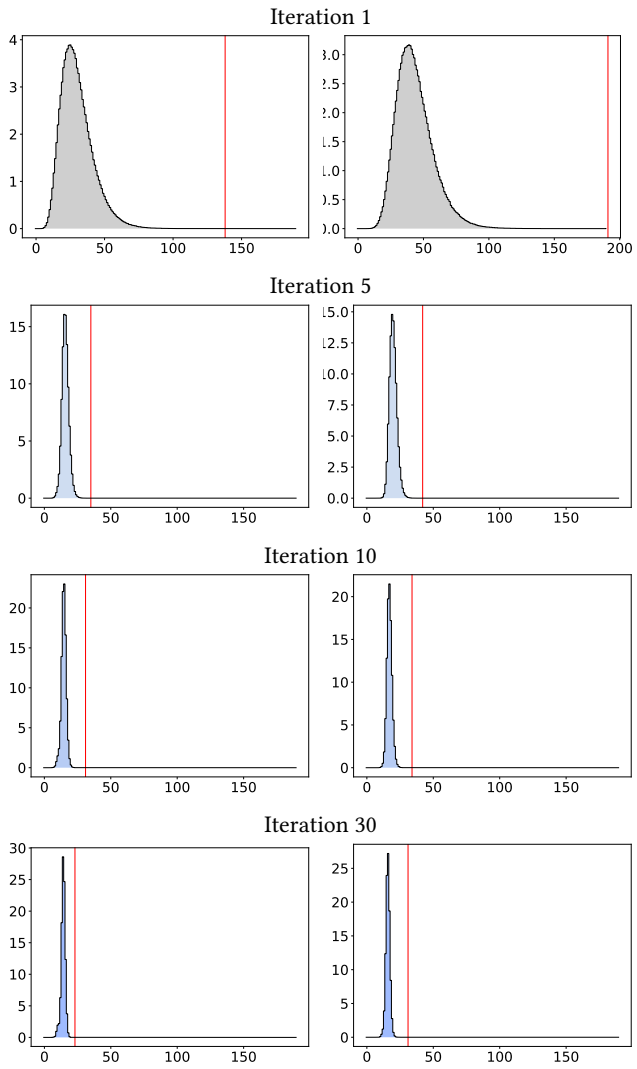

Fig. 17. Distribution of the "number of neighbors required to define the Voronoi cell" (left) and "the number of neighbors before reaching the security radius criteria" (right) at different iterations of the Lloyd algorithm. Red lines show the upper bound.

A CERTIFIED VERSION OF THE ALGORITHM

The Voronoi diagrams computed by our algorithm can be used by *meshless* methods. By meshless, we mean algorithms that solely use the geometry of the Voronoi cells, and not on the global combinatorics of the diagram, as in Lloyd relaxation, or certain types of fluid

[1] measured on an hexacore Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz. The relative performance of CGAL and GEOGRAM varies on other machines but remain similar. It is in favor of GEOGRAM for small number of cores (4), in favor of CGAL for higher number of cores with their spatial locking strategy that scales up better.

[2] We asked the authors but their implementation is not available yet, this is why more precise comparison data is not included.

simulations. We recommend using our algorithm in these cases, for which it was designed, but it may be worth it asking whether our algorithm can be made certified, in the sense that the computed Voronoi cells have *exact* and *globally consistent* combinatorics. By globally consistent, we mean that the choices taken by the algorithm in the case of co-spherical points are the same for neighboring cells, that is, not like in Figure 3. Such a certified algorithm can be used to compute the Delaunay triangulation, by outputting all the indices of the bisector planes that correspond to the Voronoi vertices (and they will be then globally coherent).

It is possible to make our algorithm certified by making a simple change in it: the only combinatorial decisions taken by our algorithm depend on classifying a Voronoi vertex with respect to a new clipping plane. As detailed below, this corresponds to the classical `in_sphere` predicate, that can be implemented using standard techniques (arithmetic filters on the GPU and arbitrary precision arithmetic fallback on the CPU). The certified version of the code is also available in the supplemental material.

A.1 Classifying Voronoi vertices and the `in_sphere` predicate

We now recall the classical relation between the clipping operation in Algorithm 3 and the `in_sphere` predicate.

Suppose you compute the Voronoi cell of vertex $\mathbf{x}_i$ using Algorithm 3, and consider the following snippet that classifies a Voronoi vertex with respect to the current clipping plane p (line 4 of the Algorithm 3):

$$(x, y, z) \leftarrow \text{IntersectPlanes}(\mathcal{P}_u, \mathcal{P}_v, \mathcal{P}_w);$$

Note that the three planes $\mathcal{P}_u, \mathcal{P}_v, \mathcal{P}_w$ that define the Voronoi vertex are the three bisectors $\Pi(\mathbf{x}_i, \mathbf{x}_u), \Pi(\mathbf{x}_i, \mathbf{x}_v), \Pi(\mathbf{x}_i, \mathbf{x}_w)$, defined by: $\Pi(\mathbf{x}_i, \mathbf{x}_u) = \{\mathbf{x} \mid d(\mathbf{x}, \mathbf{x}_i) = d(\mathbf{x}, \mathbf{x}_u)\}$ (resp. v, w), where $d(.,.)$ denotes the Euclidean distance.

The plane p is the bisector defined by another point, say $\mathbf{x}_k$:

$$p = \Pi(\mathbf{x}_i, \mathbf{x}_k) = \{\mathbf{x} \mid d(\mathbf{x}, \mathbf{x}_i) = d(\mathbf{x}, \mathbf{x}_k)\}$$

In the snippet of Algorithm 3, the Voronoi vertex $(x, y, z) = \Pi(\mathbf{x}_i, \mathbf{x}_u) \cap \Pi(\mathbf{x}_i, \mathbf{x}_v) \cap \Pi(\mathbf{x}_i, \mathbf{x}_w)$ is clipped-out if $(x, y, z, 1) \cdot p > 0$, that is if the Voronoi vertex (x, y, z) is nearer to $\mathbf{x}_k$ than $\mathbf{x}_i$.

Now remember that the Voronoi vertex (let us call it $\mathbf{m}$) is at the intersection of three bisectors:

$$(x, y, z) = \mathbf{m} = \Pi(\mathbf{x}_i, \mathbf{x}_u) \cap \Pi(\mathbf{x}_i, \mathbf{x}_v) \cap \Pi(\mathbf{x}_i, \mathbf{x}_w).$$

Thus, by the definition of the bisectors, we have that: $d(\mathbf{m}, \mathbf{x}_i) = d(\mathbf{m}, \mathbf{x}_u) = d(\mathbf{m}, \mathbf{x}_v) = d(\mathbf{m}, \mathbf{x}_w) = R$, where R denotes the radius of the circumscribed sphere of $(\mathbf{x}_i, \mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_w)$.

Now remember that $\mathbf{m}$ is clipped-out by the current clipping plane $p = \Pi(\mathbf{x}_i, \mathbf{x}_k)$ if $d(\mathbf{x}_i, \mathbf{m})(= R)$ is larger than $d(\mathbf{x}_k, \mathbf{m})$. In other words, $\mathbf{m}$ is clipped-out by $\Pi(\mathbf{x}_i, \mathbf{x}_k)$ if $\mathbf{x}_k$ is in the circumscribed sphere of $(\mathbf{x}_i, \mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_w)$.

To summarize, all the combinatorial decisions taken by the clipping algorithm depend on the `in_sphere` predicate.

A.2 Hybrid GPU-CPU `in_sphere` predicate

It is possible to robustly implement the `in_sphere` predicate by combining arithmetic filter [Melquiond and Pion 2007], exact precision arithmetics [Shewchuk 1996] and symbolic perturbation [Edelsbrunner and Mücke 1994] (see [Lévy 2016] for a tutorial). The first step (arithmetic filter) can be completely implemented on the GPU: the `in_sphere` predicate corresponds to the sign of a determinant. The question is whether the sign of this determinant is correct when computed using (limited precision) floating point arithmetics. One can pre-compute a bound, and prove that whenever the absolute value of the computed determinant is larger than this bound, then the sign is correct [Melquiond and Pion 2007]. The associated code is simple and can be implemented on the GPU. The companion source-code includes the arithmetic filter with the bound computed for both single-precision and double-precision arithmetics, as well as the program (inspired by CGAL) that computes these bounds. Whenever this happens, we use the CPU fallback, that uses the two other stages of arithmetics (arbitrary precision using expansions [Shewchuk 1996] and "simulation of simplicity" to take coherent decisions when points are co-spherical [Edelsbrunner and Mücke 1994].

To summarize, the only required change in the algorithm to make it certified is when classifying Voronoi vertices: the absolute value of the computed determinant is compared with a fixed bound. Whenever it is smaller, an error status is returned, and the cell is recomputed by the CPU fallback.

In our empirical experiment, a very large proportion of the computed determinants provably have the correct signs: for our 10 million blue-noise point tests, the sign could not be validated 1 time using double-precision arithmetics, and 8382 times using single-precision arithmetics. Clearly double precision is interesting to use in this case. However, on the GPU, the cost of double precision versus single precision highly depends on the hardware: with the V100 that has efficient double-precision FPUs, the cost is negligible (less than 10%), but with the GTX1080, this nearly doubles the overall timing. One could imagine using some form of cascading precision [Shewchuk 1996] to gain efficiency on hardware without efficient double-precision FPUs.

REFERENCES

Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. 2017. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Trans. Graph.* 36, 4 (2017), 140:1–140:12.

Shostko Alexander and Löhner Rainald. 1995. Three-dimensional parallel unstructured grid generation. *Internat. J. Numer. Methods Engrg.* 38, 6 (1995), 905–925.

Nina Amenta, Sunghee Choi, Tamal K. Dey, and N. Leekha. 2002. A Simple Algorithm for Homeomorphic Surface Reconstruction. *Int. J. Comput. Geometry Appl.* 12, 1-2 (2002), 125–141.

Nina Amenta, Sunghee Choi, and Günter Rote. 2003. Incremental constructions con BRIO. In *Proceedings of the 19th ACM Symposium on Computational Geometry, San Diego, CA, USA, June 8-10, 2003.* 211–219. https://doi.org/10.1145/777792.777824

Ricardo J. Barrientos, Fabricio Millaguir, José L. Sánchez, and Enrique Arias. 2017. GPU-based Exhaustive Algorithms Processing kNN Queries. *J. Supercomput.* 73, 10 (Oct. 2017), 4611–4634. https://doi.org/10.1007/s11227-017-2110-y

Vincente H.F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. 2010. Parallel geometric algorithms for multi-core computers. *Internat. J. Numer. Methods Engrg.* 43, 8 (2010), 663–677.

Dobrina Boltcheva and Bruno Lévy. 2017. Surface reconstruction by computing restricted Voronoi cells in parallel. *Computer-Aided Design* 90 (2017), 123–134.

Adrian Bowyer. 1981. Computing Dirichlet Tessellations. *Comput. J.* 24, 2 (1981), 162–166. https://doi.org/10.1093/comjnl/24.2.162

Y. Brenier, U. Frisch, M. Henon, G. Loeper, S. Matarrese, R. Mohayaee, and A. Sobolevskii. 2003. Reconstruction of the early Universe as a convex optimization problem. *arXiv* (September 2003). https://arxiv.org/abs/astro-ph/0304214 arXiv:astro-ph/0304214v3.

Tyson Brochu, Christopher Batty, and Robert Bridson. 2010. Matching fluid simulation elements to surface geometry and topology. *ACM Trans. Graph.* 29, 4 (2010), 47:1–47:9.

Thanh-Tung Cao. 2014. Fundamental Computational Geometry on the GPU.

Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, and Tiow Seng Tan. 2014. A GPU accelerated algorithm for 3D Delaunay triangulation. In *Symposium on Interactive 3D Graphics and Games, I3D '14, San Francisco, CA, USA - March 14-16, 2014*. 47–54. https://doi.org/10.1145/2556700.2556710

Nicolas Cuntz and Andreas Kolb. 2007. Fast Hierarchical 3D Distance Transforms on the GPU. In *EG Short Papers*, Paolo Cignoni and Jiri Sochor (Eds.). The Eurographics Association. https://doi.org/10.2312/egs.20071042

H. L. De Cougny and M. S Shephard. 1999. Parallel refinement and coarsening of tetrahedral meshes. *Internat. J. Numer. Methods Engrg.* 46, 7 (1999), 1101–1125.

Fernando de Goes, Corentin Wallez, Jin Huang, Dmitry Pavlov, and Mathieu Desbrun. 2015. Power particles: an incompressible fluid solver based on power diagrams. *ACM Trans. Graph.* 34, 4 (2015), 50:1–50:11. https://doi.org/10.1145/2766901

Christophe Delage and Olivier Devillers. 2018. Spatial Sorting. In *CGAL User and Reference Manual* (4.12.1 ed.). CGAL Editorial Board. https://doc.cgal.org/4.12.1/Manual/packages.html#PkgSpatialSortingSummary

Qiang Du, Vance Faber, and Max Gunzburger. 1999. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Rev.* 41, 4 (Dec. 1999), 637–676. https://doi.org/10.1137/S0036144599352836

Herbert Edelsbrunner and Ernst P. Mücke. 1994. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *CoRR* abs/math/9410209 (1994). arXiv:math/9410209 http://arxiv.org/abs/math/9410209

Yun Fei, Guodong Rong, Bin Wang, and Wenping Wang. 2014. Parallel L-BFGS-B algorithm on GPU. *Computers & Graphics* 40 (2014), 1 – 9. https://doi.org/10.1016/j.cag.2014.01.002

Ian Fischer and Craig Gotsman. 2006. Fast Approximation of High-Order Voronoi Diagrams and Distance Transforms on the GPU. *J. Graphics Tools* 11 (2006), 39–60.

Thomas O. Gallouët and Quentin Mérigot. 2017. A Lagrangian Scheme à la Brenier for the Incompressible Euler Equations. *Foundations of Computational Mathematics* (23 May 2017). https://doi.org/10.1007/s10208-017-9355-y

V. Garcia, E. Debreuve, and M. Barlaud. 2008. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 1–6. https://doi.org/10.1109/CVPRW.2008.4563100

V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. 2010. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*. 3757–3760. https://doi.org/10.1109/ICIP.2010.5654017

R.E. González. 2016. PARAVT: Parallel Voronoi tessellation code. *Astronomy and Computing* 17 (2016), 80 – 85. https://doi.org/10.1016/j.ascom.2016.06.003

RC Hoetzlein. 2014. Fast fixed-radius nearest neighbors: interactive million-particle fluids. In *GPU Technology Conference*. 18.

Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. 1999. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 277–286. https://doi.org/10.1145/311535.311567

project ALICE-GRAPHYS Inria. 2018. Geogram: a programming library of geometric algorithms. http://alice.loria.fr/software/geogram/doc/html/index.html.

Bruno Lévy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Computer-Aided Design* 72 (2016), 3–12. https://doi.org/10.1016/j.cad.2015.10.004

Bruno Lévy and Nicolas Bonneel. 2013. Variational Anisotropic Surface Meshing with Voronoi Parallel Linear Enumeration. In *Proceedings of the 21st International Meshing Roundtable*, Xiangmin Jiao and Jean-Christophe Weill (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 349–366.

Bruno Lévy and Yang Liu. 2010. L_p Centroidal Voronoi Tessellation and its applications. *ACM Trans. Graph.* 29, 4 (2010), 119:1–119:11.

Shengren Li and Nina Amenta. 2015. Brute-Force k-Nearest Neighbors Search on the GPU. In *Proceedings of the 8th International Conference on Similarity Search and Applications - Volume 9371 (SISAP 2015)*. Springer-Verlag, Berlin, Heidelberg, 259–270. https://doi.org/10.1007/978-3-319-25087-8_25

Yang Liu, Wenping Wang, Bruno Lévy, Feng Sun, Dong-Ming Yan, Lin Lu, and Chenglei Yang. 2009. On centroidal voronoi tessellation - energy smoothness and fast computation. *ACM Trans. Graph.* 28, 4 (2009), 101:1–101:17. https://doi.org/10.1145/1559755.1559758

Stuart P. Lloyd. 1982. Least squares quantization in pcm. *IEEE Transactions on Information Theory* 28 (1982), 129–137.

Celestin Marot, Jeanne Pellerin, and Jean-Francois Remacle. 2018. One Machine, One minute, Three billion tetrahedra. https://arxiv.org/abs/1805.08831.

Guillaume Melquiond and Sylvain Pion. 2007. Formally certified floating-point filters for homogeneous geometric predicates. *ITA* 41, 1 (2007), 57–69. https://doi.org/10.1051/ita:2007005

Chrisochoides Nikos and Nave Damian. 2003. Parallel Delaunay mesh generation kernel. *Internat. J. Numer. Methods Engrg.* 58, 2 (2003), 161–176.

Tom Peterka, Dmitriy Morozov, and Carolyn Phillips. 2014. High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 997–1007.

Jean-François Remacle. 2017. A two-level multithreaded Delaunay kernel. *Computer-Aided Design* 85 (2017), 2–9.

G. Rong, Y. Liu, W. Wang, X. Yin, D. Gu, and X. Guo. 2011. GPU-Assisted Computation of Centroidal Voronoi Tessellation. *IEEE Transactions on Visualization and Computer Graphics* 17, 3 (March 2011), 345–356. https://doi.org/10.1109/TVCG.2010.53

Guodong Rong and Tiow-Seng Tan. 2006. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D '06)*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/1111411.1111431

Chris Rycroft. 2009. Voro++: A Three-Dimensional Voronoi Cell Library in C++. 19 (12 2009), 041111.

Jens Schneider, Martin Kraus, and Rüdiger Westermann. 2010. GPU-Based Euclidean Distance Transforms and Their Application to Volume Rendering. In *Computer Vision, Imaging and Computer Graphics. Theory and Applications*, AlpeshKumar Ranchordas, João Madeiras Pereira, Hélder J. Araújo, and João Manuel R. S. Tavares (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–228.

Jonathan Richard Shewchuk. 1996. Robust Adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry, Philadelphia, PA, USA, May 24-26, 1996*. 141–150. https://doi.org/10.1145/237218.237337

C. Sigg, R. Peikert, and M. Gross. 2003. Signed distance transform using graphics hardware. In *IEEE Visualization, 2003. VIS 2003.* 83–90. https://doi.org/10.1109/VISUAL.2003.1250358

Funshing Sin, Adam W. Bargteil, and Jessica K. Hodgins. 2009. A Point-based Method for Animating Incompressible Flow. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '09)*. ACM, New York, NY, USA, 247–255. https://doi.org/10.1145/1599470.1599502

Volker Springel. 2011. Moving-mesh hydrodynamics with the AREPO code. *Computational Star Formation* (2011).

Avneesh Sud, Naga Govindaraju, Russell Gayle, and Dinesh Manocha. 2006. Interactive 3D Distance Field Computation Using Linear Factorization. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D '06)*. ACM, New York, NY, USA, 117–124. https://doi.org/10.1145/1111411.1111432

Avneesh Sud, Naga Govindaraju, and Dinesh Manocha. 2005. Interactive computation of discrete generalized voronoi diagrams using range culling. In *In Proceedings of the international symposium on Voronoi diagrams in science and engineering*.

Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. 2004. DiFi: Fast 3D Distance Field Computation Using Graphics Hardware. *Computer Graphics Forum* 23, 3 (2004), 557–566. https://doi.org/10.1111/j.1467-8659.2004.00787.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2004.00787.x

The CGAL Project. 2018. *CGAL User and Reference Manual* (4.12.1 ed.). CGAL Editorial Board. https://doc.cgal.org/4.12.1/Manual/packages.html

David Watson. 1981. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Comput. J.* 24, 2 (1981), 167–172.

Simon D. M. White and Volker Springel. 1999. Fitting the universe on a supercomputer. *Computing in Science and Engineering* 1, 2 (1999), 36–45.