Pacific Graphics (2017)
J. Barbic, W.-C. Lin, and O. Sorkine-Hornung (Editors)

Computing Restricted Voronoi Diagram on Graphics Hardware

Jiawei Han[1] Dong-Ming Yan[2] Lili Wang[†1] Qinping Zhao[1]

[1]State Key Laboratory of Virtual Reality Technology and Systems, School of Computer Science and Engineering, Beihang University
[2]National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences, China

Abstract

The 3D restricted Voronoi diagram (RVD), defined as the intersection of the 3D Voronoi diagram of a pointset with a mesh surface, has many applications in geometry processing. There exist several CPU algorithms for computing RVDs. However, such algorithms still cannot compute RVDs in realtime. In this short paper, we propose an efficient algorithm for computing RVDs on graphics hardware. We demonstrate the robustness and the efficiency of the proposed GPU algorithm by applying it to surface remeshing based on centroidal Voronoi tessellation.

1. Introduction

The Voronoi diagram is ubiquitous in nature and science. In many graphics applications, one has to compute Voronoi diagrams on given manifolds, usually represented by triangle meshes. This geometric structure is called *Restricted Voronoi Diagram* (RVD), which is defined as the intersection of a 3D Voronoi diagram and a mesh surface [ES97]. The RVD computation is exhaustively involved in surface remeshing based on *Centroidal Voronoi Tessellation* (CVT). Fig. 1 illustrates the typical process of the CVT framework.

There exist several efficient versions of RVD implementation [YLL*09, LB12, YBZW14], all on CPUs. It is still inefficient for applications where real-time remeshing is desired, since RVD computation is invoked repeatedly in remeshing process. To further improve the efficiency, we propose a GPU implementation of the RVD algorithm, which is approximately one order of magnitude faster than current CPU implementations.

2. Related Work

Yan et al. [YLL*09] first propose a practical algorithm to compute the exact RVD on a triangle mesh for a given set of points. They have to build a kd-tree and construct the 3D Voronoi diagram of the point set during the computation. To accelerate this process, Lévy and Bonnel [LB12] drop the requirement of explicitly constructing the 3D Voronoi diagram, which is the most time consuming component in [YLL*09]. In their work [LB12], only a kd-tree is used to dynamically access the incident bisectors of each triangle, which avoids the explicit construction of the 3D Voronoi diagram. A *security radius* is used to terminate the processing of each triangle.

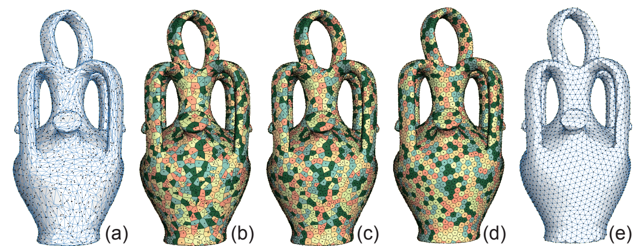

Figure 1: *Illustration of CVT optimization. (a) The Botiji model with 6k facets and 3k sampled points. (b) Restricted Voronoi diagram of the pointset. (c) Result after one iteration. (d) Result after 126 iterations. (e) The restricted Deluanay triangulation.*

Several GPU-based approaches have been developed to improve the performance of the CVT-based remeshing framework. Rong et al. [RLW*11] propose an GPU implementation of the centroidal Voronoi tessellation, where the Voronoi diagram is computed in a 2D image space using the jump flooding algorithm [RT06]. Although this approach improves the efficiency a lot, the scalability of their method is limited by the size of graphical memory, where only several thousands of points can be involved in the computation, and the input mesh has to be drastically simplified to create the geometry image representation. Later, Fei et al. [FWW14] present a GPU implementation of the L-BFGS algorithm, which is also used in CVT framework together with the RVD computation. This approach can also be used in our framework to replace the CPU counterpart for further improvement. More recently, Leung et al. [LWH*15] propose a unified framework for isotropic surface remeshing. The input surface is discretized in voxels, and the RVD is approximated by clustering neighboring voxels. This approach

† Corresponding author (wanglily@buaa.edu.cn)

© 2017 The Author(s)
Eurographics Proceedings © 2017 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/pg.20171320

can deal with only uniform remeshing, and it is not clear how it can be extended to the adaptive case. In contrast to these methods, we compute the exact RVD on GPUs directly (up to the machine precision). We do not have any limitation on the input points size, and can also be used in adaptive remeshing without involving any overhead computation.

3. Preliminaries

In this section, we review the definitions of the *Voronoi diagram*, the restricted Voronoi diagram and the centroidal Voronoi diagram.

3.1. Voronoi Diagram

Given a set of points $\mathbf{X} = (\mathbf{x}_1, ..., \mathbf{x}_n) \in \mathbb{R}^d$, the Voronoi diagram $\Omega(\mathbf{X})$ is the collection of Voronoi cells $\Omega(\mathbf{x}_i)$ (denoted as Ω_i for short), i.e., $\Omega(\mathbf{X}) = \{\Omega_i\}$. A *Voronoi cell* is defined as:

$$\Omega_i = \{\mathbf{x} \in \mathbb{R}^d | d(\mathbf{x}, \mathbf{x}_i) \leq d(\mathbf{x}, \mathbf{x}_j), \forall j\}, \quad (1)$$

where $d(\cdot, \cdot)$ denotes the Euclidean distance. The subset Ω_i is called the Voronoi cell of $\mathbf{x}_i$.

3.2. Restricted Voronoi Diagram

Given a two-manifold surface $\mathcal{S} \subset \mathbb{R}^3$, and a set of finite samples $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^n$ sampled on $\mathcal{S}$, the restricted Voronoi diagram is defined as the intersection of the 3D Voronoi diagram $\Omega = \{\Omega_i\}_{i=i}^n$ of $\mathbf{X}$ and the surface $\mathcal{S}$, as shown in Fig. 1(b-d).

$$\Omega_{|S} = \Omega \cap S = \{\Omega_i \cap S\}_{i=1}^n.$$

A *Restricted Voronoi Cell* (RVC) is defined as:

$$\Omega_{i|S} = \{\mathbf{x} \in S, d(\mathbf{x}, \mathbf{x}_i) \leq d(\mathbf{x}, x_j), \forall \mathbf{x}_j \in \mathbf{X}, j \neq i\}.$$

The dual of the RVD is a subcomplex of the 3D Delaunay triangulation, called the *Restricted Delaunay Triangulation* (RDT) (see Fig. 1(e)). If the ε-sampling property [ABK98] and the topological ball property [ES97] are met, each RVC is a single connected component and the RDT is topologically equivalent to the underlying surface $\mathcal{S}$.

3.3. Centroidal Voronoi Tessellation

The centroidal Voronoi tessellation is a special type of the Voronoi diagram, which requires the generator of each Voronoi cell coincides with its centroid. The CVT on mesh surface can be achieved by minimizing the following energy function [YLL*09]:

$$F(x) = \sum_{i=1}^{n} \int_{\Omega_{i|S}} w(\mathbf{x}) ||\mathbf{x} - \mathbf{x}_i||^2 d\mathbf{x}, \quad (2)$$

where $w(x)$ is a density function defined over the surface. We denote w_i as the weight defined at each vertex $\mathbf{v}_i$ of the input mesh. The weight is linearly interpolated within each triangle.

4. GPU-RVD Algorithm

In this section, we first introduce the basic concepts required by the latest CPU-RVD algorithm [LB12]. Then, we explain our GPU-RVD implementation and propose a method that can reduce the writing data conflict.

4.1. Preliminaries of RVD

As defined in Sec. 3.2, we have to compute the Voronoi diagram $\Omega(\mathbf{X})$, which is required by the RVD computation. However, constructing the Voronoi diagram is quite time-consuming. Since each Voronoi cell is a convex polytope, it can be defined as the intersection of half-spaces:

$$\Omega_i = \cap_{k=1}^{v_i} \prod^{+} (i, j_k),$$

where $\prod^+ (i, j_k)$ denotes the half space bounded by the bisector of $(\mathbf{x}_i, \mathbf{x}_j)$ (called $b(i, j)$ for short) that contains $\mathbf{x}_i$. Lévy and Bonnel [LB12] demonstrate that this representation of the Voronoi cell is well suited for the computation of the Voronoi cells by clipping planes (see Fig. 2 (c)). The classic Sutherland & Hodgman's reentrant clipping [SH74] can be used for clipping a polygon by convex half-planes. To compute bisectors, the vertices $\mathbf{X}$ are organized in a geometric search data structure, i.e., kd-tree, such that for any $\mathbf{x}_i$ one can efficiently retrieve the list of the nearest k neighbors $\{\mathbf{x}_j\}$ sorted by increasing distance to $\mathbf{x}_i$. These k-nearest neighbors are then used as bisectors of Voronoi planes of current point to clip incident facets of the input mesh. A safe radius is estimated for each facet-point pair, which is used to terminate the clipping process [LB12].

4.2. GPU-RVD implementation

The key to the RVD algorithm is to compute the intersection of $\Omega(\mathbf{X})$ and the surface $\mathcal{S}$, which consists of a set of triangles (or facets). Since the intersection of a Voronoi cell and a facet is independent from both the other facet and the other cell, the computation process has a great potential of parallelism.

We first propose a per facet-based parallel version of RVD computation. The pseudo code is given in Alg. 1. In the pre-process stage, we build a kd-tree to obtain two segments of data. We use a multi-core CPU parallel implementation kd-tree for this purpose. One keeps the k nearest indices of each point $\mathbf{x}_i$ in linear memory which will be used to compute the bisectors representing the Voronoi cell Ω_i. The other stores the nearest neighbor of the centroid of each facet. Since Voronoi cell of the nearest point has a non-empty intersection with the corresponding facet, we initialize an incident cell *stack* S of the facet by the index of this point.

In each iteration, if the stack S is not empty, we parallelly handle each pair of the incident cell-facet (see Fig. 2(e)). The process is showed in Alg. 2. We propose a data structure which is called *polygon*. Since the GPU programming is not allowed to allocate the memory dynamically, *polygon* statically contains n vertices ($n = 15$ in our implementation). Each vertex contains five attributes (x, y, z, w, s) which represent the position, the weight, and the opposite neighbor of current computing bisector. The *polygon* is initialized with the three vertices of current facet. During the clipping process, the *polygon* is updated by its intersection with bisectors. Once the clipping is done, the position and the weight are used to update the coordination of the $\mathbf{x}_i$ while the attribute s specifies the index of points of the next iteration since both two sides of the clipped bisector should have an intersection of the current *polygon*.

© 2017 The Author(s)
Eurographics Proceedings © 2017 The Eurographics Association.

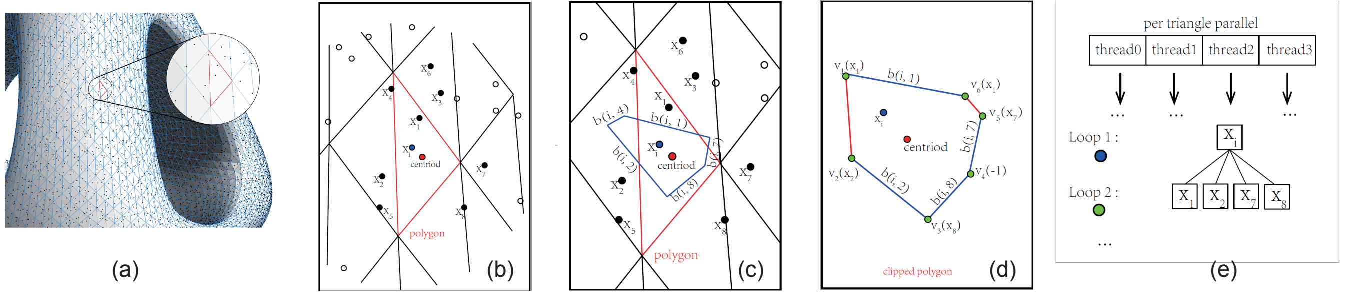

Figure 2: *(a): Choose a triangle in model Genius (13k facets). (b): Find the nearest point (blue dot) of centroid (red dot) and the neighbors of the point (black dots). (c): Compute the bisectors ($b(i,j)$) of $\mathbf{x}_i$ and its neighbors and clip with the polygon. (d): Initialize vertex's s attribute with -1. In the process of clipping, we update it by the index of the opposite point with the bisector intersects the polygon. (e): Parallel loops. In the first loop, we initialize the computing stack with the nearest point $\mathbf{x}_i$. In the later loop, we simultaneously handle the points which are stored in the clipped polygon's vertex (s property).*

Algorithm 1 compute RVD (per triangle parallel)

Input: a point set $\mathbf{X}$; a triangle mesh $\mathcal{M}$;
Output: $\Omega \cap \mathcal{M}$
find k nearest neighbors for each point $\mathbf{x}_i \in \mathbf{X}$
find the nearest neighbor for each triangle facet of $\mathcal{M}$
for each $t \in \mathcal{M}$ **in parallel do**
  update stack S from the preserved nearest neighbor
  while $S \neq \emptyset$ **do**
    for each $\mathbf{x}_i \in S$ **in parallel do**
      clip the current triangle t with $\Omega(\mathbf{x}_i)$
      for each vertex v in clipped polygon **do**
        if the label index has not been processed **then**
          add the index into S
        end if
      end for
      atomic update location of point $\mathbf{x}_i$
    end for
  end while
end for

4.3. Parallel strategy and conflict optimization

Not only the facets are independent to each other but also the cells. We can also create one thread per cell for parallelization. Although this strategy shows lower performance in most conditions as the index-search of this method causes more bandwidth of data transmission in GPU, when a triangle has more than 20 points to process, the operations of static stacks in per-facet strategy will greatly slow down the speed. Then we will use per-cell strategy as an alternative.

To store the RVD data in each thread of the processing units, we have to use atomic operations to ensure the correctness of our algorithm. However, when thousands of threads are trying to access a segment of memory locations, a great deal of contention for our bins can occur. To address this issue, we split our atomic operations into two phases. In phase 1, each parallel block will be assigned a segment of shared-memory to store the temporary data. In phase

Algorithm 2 clip a facet by $\Omega(\mathbf{x}_i)$

Input: $\mathbf{x}_i$; its neighbors' index $p[k]$; a triangle facet t;
Output: clipped polygon ρ
initialize the ρ from triangle facet t
for each neighbor j from $p[0]$ to $p[k-1]$ **do**
  compute the bisector b of the $\mathbf{x}_i$ and $p[j]$
  if ρ intersects the bisector b **then**
    clip the triangle facet into a polygon ρ
    record the index j into the vertex of the intersection
  end if
end for
Return ρ

2, after all the threads finish the tasks, we merge the data into the global memory and copy them into CPU for the next iteration. The atomic operations only consume about 10% percentages of the time if we use this optimization, while it will take up to 20% if not. This strategy greatly reduces the contention and improves the writing performance.

5. Experimental Results

We test our algorithm on a PC with an Intel 4.00GHz and 8GB memory. The graphics card is an Nvidia GTX 1070 with 1920 CUDA cores and 4.0GB GPU memory. Our program is compiled using CUDA version 8.0.

Performance. We first evaluate the performance of our GPU-RVD computation algorithm. As shown in Fig. 3 (left), we use the Bunny model (69k facets) as input mesh and test our algorithm with an increasing number of points, from 20 to 10^6, sampled on the surface. From the timing curves, our parallel algorithm outperforms the existing CPU algorithm in one order of magnitude. Fig. 3 (right) shows the performance of our parallel algorithm and the serial algorithm when we fix the number of points while increasing the mesh resolution from 3k to 100k.

The time complexity of our algorithm is $O(log_m)$ in per-facet

© 2017 The Author(s)
Eurographics Proceedings © 2017 The Eurographics Association.

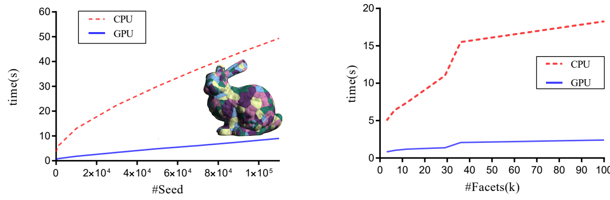

Figure 3: *Left: Running time comparison between our parallel algorithm and linear algorithm on model Bunny (69k facets). Right: Fixing the number of points (10k), increase the mesh resolution.*

strategy and $O(n/m log_m)$ in per-cell strategy while the CPU-version is $O(n log_m)$. The space complexity is $O(log_m)$, where n and m represent the number of triangles and samples, respectively.

Comparison. We compare the performance with [LWH*15] (Fig. 4(left)) using the same input mesh with different number of samples. Our algorithm is roughly 3∼5 times faster. Moreover, the RVD in their method is approximated by clustering voxels while our approach computes the exact version. Another difference is that our method works well for adaptive CVT, which is a limitation of theirs. Fig. 4(right) shows the performance comparison of our algorithm with [RLW*11]. Ours is approximately 3∼7 times faster. The key differences between our approach and theirs are two-folds. First, our approach does not involve any parameterization, leading to better remeshing quality. Second, we do not have any limitation on the number of samples, since all the computation is performed on-the-fly, while their method is limited by the graphical memory.

Models(facets)	Points	GPU	CPU		Multi-core CPU	
		time	time	speedup	time	speedup
Davidhead(24k)	1.5e4	5.9	77.6	13.1	18.4	3.1
Gargoyle(60k)	2e4	9.1	141.2	15.1	31.1	3.4
Dragon(100k)	5e4	24.9	308.9	12.4	64.3	2.6

Table 1: *The running time(ms) of per iteration of the computation of RVD with GPU, CPU, Multi-CPU algorithm.*

Table 1 shows the timing comparison with the CPU algorithm and the multi-core CPU algorithm. For each input model, we choose a proper number of seeds which is comparable to the vertex number of each model. Then we record the average time of one iteration of the RVD computation. Our algorithm is 8∼15 times faster than the CPU algorithm and 2∼4 times faster than the multi-core CPU algorithm.

6. Conclusion and Future Work

We have presented a simple yet efficient algorithm to compute the restricted Voronoi Diagram on a mesh surface. Our approach significantly speeds up over its CPU counterparts without loss of quality, especially in the high-resolution models.

The main advantage of our approach over the CPU version is the computation efficiency. However, our algorithm is not well suited for the pointset with varying number of points in different iterations, since the GPU memory size is assigned statically. Furthermore, although we use the lightweight data structure in GPU, the data transmission is still the main factor that limits the algorithm performance. In the future, we would like to exploit more strategies to optimize this part and accelerate the parallel algorithm.

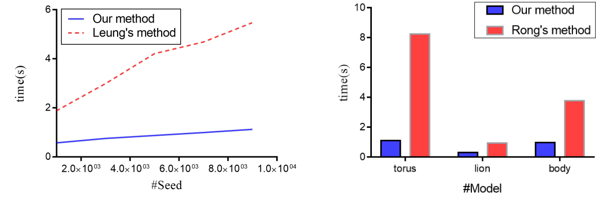

Figure 4: *Comparisons of our method and other similar GPU-CVT methods. Left: the Dragon model with 50k facets and different samples from 1k to 9k. Right:Three models with 8k, 10 and 28k facets respectively. Computing CVT with 1k sites using 500, 135 and 262 iterations, our method shows better performance in speed.*

Acknowledgments

We would like to thank the authors of [LWH*15] for providing us with their software for comparison. This work is partially funded by the NSFC (61772051, 61772523, 61372168, 61620106003, 61272349, and 61190121).

References

[ABK98] AMENTA N., BERN M., KAMVYSSELIS M.: A new Voronoi-based surface reconstruction algorithm. In *Proc. ACM SIGGRAPH* (1998), pp. 415–421. 2

[ES97] EDELSBRUNNER H., SHAH N. R.: Triangulating topological spaces. *IJCGA 7*, 4 (1997), 365–378. 1, 2

[FWW14] FEI Y., WANG W., WANG B.: Parallelize l-bfgs-b on the gpu. *Computers & Graphics 40*, 1-9 (2014). 1

[LB12] LÉVY B., BONNEEL N.: Variational anisotropic surface meshing with Voronoi parallel linear enumeration. In *Proc. of the 21st IMR* (2012), pp. 349–366. 1, 2

[LWH*15] LEUNG Y.-S., WANG X., HE Y., LIU Y.-J., WANG C. C.-L.: A unified framework for isotropic meshing based on narrowband euclidean distance transformation. *Comput. Vis. Media 1*, 239-251 (2015). 1, 4

[RLW*11] RONG G., LIU Y., WANG W., YIN X., GU X., GUO X.: GPU-assisted computation of centroidal Voronoi tessellation. *IEEE Trans. on Vis. and Comp. Graphics 17*, 3 (2011), 345–356. 1, 4

[RT06] RONG G., TAN T. S.: Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *I3D* (2006), pp. 109–116. 1

[SH74] SUTHERLAND E., HODGMENT W.: Reentrant polygon clipping. *Communications of the ACM 17*, 1 (1974). 2

[YBZW14] YAN D.-M., BAO G., ZHANG X., WONKA P.: Low-resolution remeshing using the localized restricted Voronoi diagram. *IEEE Trans. on Vis. and Comp. Graphics 20*, 10 (2014), 418–1427. 1

[YLL*09] YAN D.-M., LÉVY B., LIU Y., SUN F., WANG W.: Isotropic remeshing with fast and exact computation of restricted Voronoi diagram. *Computer Graphics Forum (Proc. SGP) 28*, 5 (2009), 1445–1454. 1, 2

© 2017 The Author(s)
Eurographics Proceedings © 2017 The Eurographics Association.