Restricted Power Diagrams on the GPU

Justine Basselin, Laurent Alonso, Nicolas Ray, Dmitry Sokolov, Sylvain Lefebvre, Bruno Lévy

Restricted Power Diagrams on the GPU

J. Basselin, L. Alonso, N. Ray, D. Sokolov, S. Lefebvre, B. Lévy

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
firstname.lastname@inria.fr

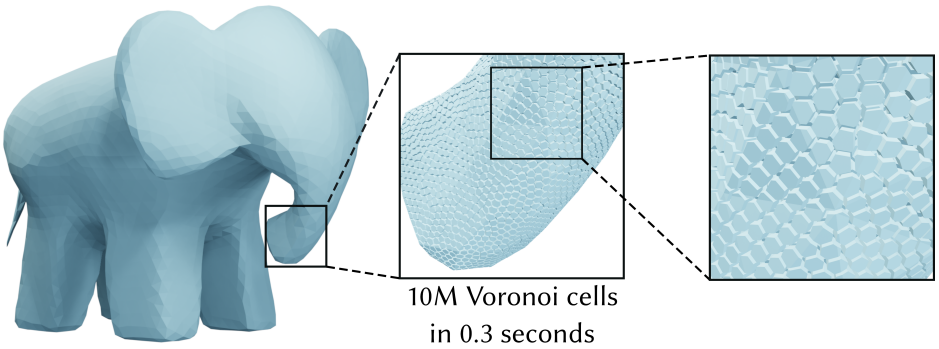

Figure 1: *We compute 3D Voronoi diagrams of 10 million points restricted to a mesh in 300 ms on the GPU (Nvidia V100).*

Abstract

We propose a method to simultaneously decompose a 3D object into power diagram cells and to integrate given functions in each of the obtained simple regions. We offer a novel, highly parallel algorithm that lends itself to an efficient GPU implementation. It is optimized for algorithms that need to compute many decompositions, for instance, centroidal Voronoi tesselation algorithms and incompressible fluid dynamics simulations.

We propose an efficient solution that directly evaluates the integrals over every cell without computing the power diagram explicitly and without intersecting it with a tetrahedralization of the domain. Most computations are performed on the fly, without storing the power diagram. We manipulate a triangulation of the boundary of the domain (instead of tetrahedralizing the domain) to speed up the process. Moreover, the cells are treated independently one from another, making it possible to trivially scale up on a parallel architecture.

Despite recent Voronoi diagram generation methods optimized for the GPU, computing integrals over restricted power diagrams still poses significant challenges; the restriction to a complex simulation domain is difficult and likely to be slow. It is not trivial to determine when a cell of a power diagram is completely computed, and the resulting integrals (e.g. the weighted Laplacian operator matrix) do not fit into fast (shared) GPU memory. We address all these issues and boost the performance of the state-of-the-art algorithms by a factor 2 to 3 for (unrestricted) Voronoi diagrams and a $\times 50$ speed-up with respect to CPU implementations for restricted power diagrams. An essential ingredient to achieve this is our new scheduling strategy that allows us to treat each Voronoi/power diagram cell with optimal settings and to benefit from the fast memory.

CCS Concepts

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

Introduction

When numerical optimizations involve complex geometric objects, it is often necessary to discretize them. Voronoi diagrams are interesting for this purpose as tools to discretize geometric objects because they are controlled by a set of points (seeds). They can be exploited, for instance, to simulate diffusion processes (Centroidal Voronoi Tessellations/Lloyd) [Wan17], or to model random yet well-distributed foams [MDL16].

The article was accepted at Eurographics 2021 and is pending publication.

Power diagrams generalize Voronoi diagrams by adding a weight to each seed which allows to control the (relative) scale of each cell. Another advantage of the power diagram is its link with optimal transport: given a set of seeds and a domain Ω, there exists a set of weights such that the map that associates each seed to its power cell restricted to Ω is the optimal transport between the seeds and Ω. This property was exploited to generate blue noise [dGBOD12, XLC*16] and more recently to enforce incompressibility in fluid dynamics [dGWH*15a, GM17].

In those applications, the Voronoi/power diagrams are constructed only to compute integrals: CVT requires the volume and the barycenter of Voronoi cells and optimal transport additionally needs to support power diagrams and to compute the weighted Laplacian operator matrix [dGBOD12]. Computing those values is often a bottleneck because they are needed inside a critical loop of the optimization process. We improve the performance by more than an order of magnitude with respect to equivalent solutions by directly computing those integrals on the GPU.

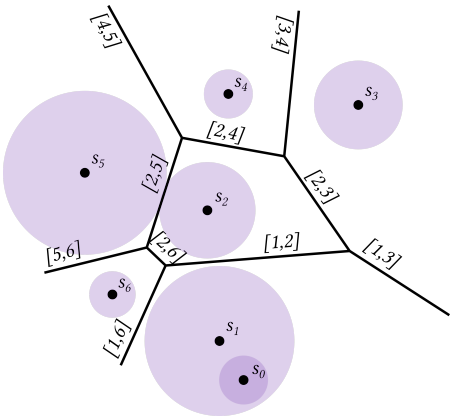

Figure 2: *In two dimensions, a cell of a power diagram can be seen as the intersection of half-planes generated by radical axes that are defined by circles centered on the seeds with radii given by the corresponding weights.*

Previous work

Computing Delaunay triangulation (or Voronoi diagram) is a fundamental task in computational geometry. In most cases, it is done by an extension of the Bowyer-Watson algorithm [Bow81, Wat81]. However, when the seed distribution is homogeneous, it is possible to explore alternative solutions that are easier to implement in parallel.

Extensions of the Bowyer-Watson algorithm:

Most algorithms extend the classical Bowyer-Watson algorithm [Bow81, Wat81] (point location and cavity re-triangulation) and rely on spatial sorting [ACR03, DD18] to speed-up the point location phase. Parallel implementations are optimized for specific settings. To compute huge meshes, it is possible to split the point set by regions and distribute them on a cluster [Ryc09, ND03, AR95, DCS99, Gon16]. When the mesh fits into the memory of a single computer, multi-core processors with tens of threads can edit the mesh simultaneously with a reasonable proportion of conflicts [The18, BMPS10, Inr18, Rem17, MPR18]. GPU implementation can be derived directly from CPU strategies [CNGT14, Cao14], but concurrent editing of a global mesh data structure does not scale-up very well for high number of threads and in particular does not reach high efficiency on modern GPU architectures.

Computing each Voronoi cell independently

A different approach is to compute each Voronoi cell independently as the intersection of a set of half-spaces. For a seed S_i, we know that all points closer to another seed S_j than to S_i do not belong to the cell. This set is defined by an half-space $ax+by+cz+d > 0$ where $ax+by+cz+d = 0$ is the bisector of S_iS_j. From this observation, we can define the Voronoi cell $\mathcal{C}_i$ as the intersections of the corresponding half-spaces.

Like the Voronoi diagram, the power diagram, also called a Laguerre–Voronoi diagram, is generated by a set of points; however, unlike the Voronoi diagram, these points are weighted, with the weights influencing the size of the cells. The power diagram is a form of generalized Voronoi diagram, and coincides with the Voronoi diagram in the case of equal weights. Like for the Voronoi diagrams, the cells of a power diagram are convex polytopes and can also be seen as the intersection of half-spaces; the only difference is that these half-spaces are not generated by bisectors, but by radical hyperplanes (refer to Fig. 2).

N.B.: note that while cells of a power diagram are convex polytopes, their restriction to a domain is not necessarily convex. Moreover, a restriction of a cell to a domain can even have multiple connected components, thus a special care must be taken to compute the integrals correctly.

To compute (unrestricted) Voronoi cells using this definition, each cell is initialized as the full space, then is iteratively clipped by half-spaces generated by the other seeds. Fortunately, it is not necessary to visit all the other seeds. To do so, we visit the seeds in order of distance to the current seed. Prior to every clipping, we compute the minimum radius of a ball centered on the seed point and containing the cell. If the distance from the seed to the currently considered neighbor is greater than twice the radius, then the bisector does not clip the cell and we can stop the process. This condition is referred to as the security radius criterion [LB13].

This approach is easy to parallelize, and it outperforms the extensions of the Bowyer-Watson algorithm in certain cases (when the point set has a homogeneous distribution). The implementation of VORO++ [Ryc09] exploits the versatility of CPU to deal with its dynamic combinatorial data structure. For a GPU implementation, it is better to focus on a more compact data structure [RSLL18]. Restriction of this latter algorithm to a tetrahedral mesh was recently done [LY19, LMGY20] by considering the intersection of each Voronoi cell with each nearby tetrahedron. The choice of the tetrahedra to intersect with the current Voronoi cell is a complex one; the authors chose a heuristic based on their respective barycenters, leading to problematic computational errors. Our algorithm is exhaustive and does not miss pairs of primitives to intersect, thus avoiding these errors. We avoid tetrahedralizaiton of the domain (it

is defined as a triangulated surface), what allows us to gain a $\times 100$ speed-up with respect to [LMGY20]. We also outperform CPU implementations for the specific case of homogeneously distributed seeds.

Note that there also exists a GPU implementation for 2D restricted Voronoi diagrams [FRWW14], but it is restricted to a triangulated domain (not a volume). Their restriction makes it possible to directly clip each triangle of the domain without explicitly computing the Voronoi cells.

Contributions

The proposed algorithm improves performances over the state of the art:

- We obtain a $\times$ 2–3 speed-up of [RSLL18]'s algorithm, thanks to our new scheduling strategy for unrestricted Voronoi diagrams.
- Our algorithm is more than 100 times faster than [LMGY20]'s GPU algorithm for integrating over a constrained domain, thanks to directly evaluating integrals over the domain, instead of intersecting cells with a tetrahedral mesh.

Moreover, it has the following features:

- The simulation domain is bounded by a triangulated surface.
- It supports large and irregular point set distributions (like white noise) while consuming less memory than previous work.
- Power diagrams are supported, albeit with some limitations on weight distribution corresponding to our observations on the simulation of incompressible fluids.
- New integrals are supported: for example, the weighted Laplacian operator that is required for semi-discrete optimal transport and fluid simulations.

1. Overview

Our algorithm (Alg. 1) takes as an input a set of weighted seeds (3D points and weights) and a domain defined by a triangulated surface delimiting its boundary. The output is a set of integrals computed over the 3D power diagram restricted by the domain. In our experiments, we have tested the integrals necessary for Lloyd's algorithm and power particles fluid simulation [dGWH*15a]; these are the volumes and the barycenters of the restricted power diagram cells, as well as the weighted Laplacian operator matrix necessary to compute a semi-discrete optimal transport.

First we precompute on the CPU an acceleration data structure to speed up local accesses to the domain's geometry (*domain_grid*, Alg. 1, line 1). It allows for a very efficient computation of integrals restricted to the domain, as explained in §4. Then we initialize on the GPU the spatial search data structure for the seeds (*seed_grid*, Alg. 1, line 2). After that, we start evaluating the integrals: we fetch the nearest neighbors of each seed (Alg. 1, line 8), then use them to compute the integrals over the restriction of the cell to the domain (Alg. 1, line 9).

To process all the seeds, we introduce a new **scheduling strategy** that optimizes both the running time and the GPU memory consumption. The overall scheduling process takes place inside two nested loops:

The outer loop (Alg. 1, line 4). The idea is to tune the algorithm differently for each pass: the very first iteration is fast, but fails to integrate over some cells of the power diagram, whereas subsequent iterations are increasingly safer (but slower) to process more complex (but fewer) cases. We place the failed seeds in a stack to process them in a subsequent pass.

More precisely, in our algorithm, a trade-off between the speed and the success rate is given by 3 parameters K, P, V introduced in [RSLL18] and detailed in §6.1. We initialize the parameters according to the seed distribution (and the weights for the power diagram) to process the majority of the seeds in the first pass. Each subsequent iteration increases the parameters K, P, V at each iteration until the integrals of all cells are correctly evaluated. We observed that simply scaling K, P, V by a factor 1.5 at each iteration was efficient in all settings (Fig. 10). Note that it is easier to evaluate the integrals for the cells that do not intersect the domain boundary (§3), than for the cells intersecting the boundary (§4). Running these cases at the same time would force executing diverging branches between threads, which is very inefficient on SIMD architectures like GPUs. Therefore, we split the first pass into two substeps. In the first substep we consider only the cells that are guaranteed to lie entirely inside or outside the domain and the second substep considers the rest of the cells. Note that for the first substep we compute all the cells, but discard those with the bounding box (*DomainGrid* in Alg. 1, line 1) intersecting the boundary of the domain.

The inner loop (Alg. 1, line 7). Computing the integrals over a cell requires first to get the nearest neighbors of the seed, second to construct the cell and to evaluate integrals. While it is possible to execute all steps in the same thread, it is more efficient to pre-compute the nearest neighbors of all seeds in a large array [RSLL18] (Alg. 1, line 8) because more threads can be launched simultaneously without forcing access to slow memory (only shared memory). The construction of the cell and evaluation of integrals are then computed from this array (Alg. 1, line 9).

A drawback of this solution (observed in [RSLL18]) is that the nearest-neighbors array does not fit into memory for some settings. This situation is getting worse with the weighted Laplacian operator matrix that consumes even more memory. Our solution is to process the seeds in batches that always fit into the memory. The number of seeds per batch is determined by the parameter K and the memory available on the GPU (Alg. 1, line 5).

The rest of the paper is organized as follows: first we revisit the nearest neighbors query (§2.1) and Voronoi cell computation (§2.2) as done in [RSLL18]. Then we show how to support power diagrams (§2.3) and compute the weighted Laplacian operator matrix (§3). Then we show how the integrals can be restricted to the simulation domain (§4). Finally, §6 presents performances of our algorithm in representative use cases and discusses the limitations.

2. Unrestricted power diagram computation

In this section we show how to compute unrestricted power diagrams. First we revisit the case of Voronoi diagrams obtained by computing each cell (§2.2) from its neighbor seeds (§2.1). Then we show how to extend it to power diagrams (§2.3).

Algorithm 1: Overview

Input: float4 seeds[#*seeds*]; // *seeds: coordinates and weights*
Input: TriangleMesh $\partial\Omega$; // *boundary domain*
Input: int K, P, V; // *initial algorithm settings*
Output: float4 result[#*seeds*]; // *integrals (volume, barycenter, weighted Laplacian etc.)*

1 $dg \leftarrow domain\_grid(\partial\Omega)$; // §4
2 $sg \leftarrow seed\_grid(seeds)$; // §2.1
3 $to\_process \leftarrow \{1,..,\#seeds\}$;
4 **while** $to\_process \neq \emptyset$ **do**
5 int $s \leftarrow batchsize(K)$;
6 $failed \leftarrow \emptyset$;
7 **for** $batch \in split(to\_process, s)$ **do**
8 int $knn[s][k] \leftarrow$ get_knn(sg, $batch$); // §2.1
9 result.update(dg, $batch$, knn, $failed$); // §2.2, §4
10 $(K,P,V) \leftarrow 1.5(K,P,V)$;
11 $to\_process \leftarrow failed$;
12 sg.permute(*result*); // *Cancel the re-ordering done in §2.1*

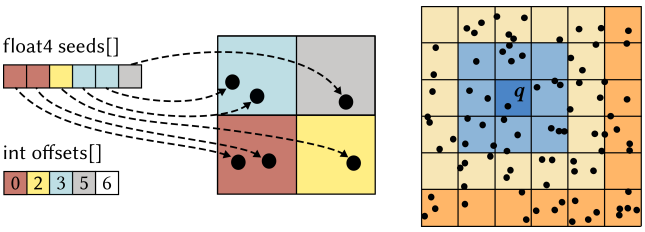

Figure 3: *Left: in order to retrieve quickly all points inside a voxel, we sort the point-set by corresponding voxel id. Right: to find k nearest neighbors for a query point $\mathcal{S}$ located in the dark blue voxel, we visit all neighboring voxels in concentric rings.*

2.1. *k*-NN query

To compute a Voronoi cell of a given seed $\mathcal{S}$, we need to find its k nearest neighbors. To do so, we use an algorithm [RSLL18] that is based on two ideas (Fig. 3):

- If we suppose a homogeneous seed distribution in space, it is possible to have an efficient spatial search structure that we call *seed_grid* (Alg. 1–line 2). The idea is to define a regular (voxel) grid in space, so we can efficiently visit all the seeds in the given voxel. In practice, it is done by reordering the points and computing a voxel offsets array. We also need an indirection map to restore the original order of the seeds.
- Equipped with the spatial search structure, we can find the k-nearest neighbors of a given seed $\mathcal{S}$. The idea is to get the seeds from the voxel that contains $\mathcal{S}$, then visit neighboring voxels in concentric rings until we are guaranteed to obtain the k nearest neighbors. The search is stopped when the closest unvisited ring is further than k neighbors we have already found. In this algorithm, the neighboring points array is sorted by the distance to the seed $\mathcal{S}$.

The difference of our implementation with [RSLL18] is the way to perform our requests: instead of requesting the k nearest neighbors for all seeds at once, we perform several smaller requests. It allows to request different numbers of neighbors (outer loop) to better fit the difficulty of computing each cell and to keep the result stored in fast shared memory (inner loop).

2.2. Convex cells

A Voronoi (power diagram) cell is a convex polyhedron that can be seen as an intersection of a number of halfspaces. To represent a convex polyhedron, we use the data structure introduced in [RSLL18]. The data structure is compact and well-suited for a GPU implementation: it consists of an array of halfspace equations (float4) and an array of triplets of integers representing the polyhedron vertices. Each triplet stores the indices of three halfspace equations incident to the vertex, enumerated in clockwise order.

Following [RSLL18], to compute a Voronoi cell of the seed $\mathcal{S}$, we initialize it as the bounding box of all the seeds. Then, we visit the neighboring seeds and iteratively clip the cell by the corresponding bisector halfspaces. Since the neighbors are sorted by the distance to the seed $\mathcal{S}$, we can stop the clipping process when the security radius criterion is met, i.e. the closest unvisited neighbor is further to $\mathcal{S}$ than twice the maximum distance of cell vertices to $\mathcal{S}$ (then the bisector is guaranteed to not intersect the cell).

The key routine here is the clipping of the convex cell by a halfspace, it can be done in 3 steps (Fig. 4):

1. detect vertices to be removed by simply evaluating the new halfspace equation on each vertex,
2. iteratively remove them from the array of vertices. They are removed in an order that allows to maintain a representation of the hole (Fig. 4–red arrows) produced in the mesh by removing vertices. This representation is a circular list of halfplane indices (stored in an array) such that two consecutive elements defines a dangling edge.
3. add the new halfspace equation to the array and add a new vertex for each dangling edge, the new vertex being simply the triplet composed by the new halfspace id and two consecutive halfplane ids in the circular list.

This algorithm is efficient on the GPU because it manipulates static arrays only and the relations between elements are easy to update: they are directly encoded in the representation of the vertices.

2.3. Power diagram extension

A power diagram can be computed as an intersection of halfspaces just like a Voronoi diagram. However, the halfspaces are now delimited by the chordale equation [Aur87]: the point q is on the chordale of seeds $\mathcal{S}_i$ and $\mathcal{S}_j$ with respecting weights $w(\mathcal{S}_i)$ and $w(\mathcal{S}_j)$ if $2q \cdot (\mathcal{S}_j - \mathcal{S}_i) = w(\mathcal{S}_i) - w(\mathcal{S}_j) + \|\mathcal{S}_j\|^2 - \|\mathcal{S}_i\|^2$.

Changing the halfspace equation implies that the security radius criterion is not sufficient anymore to guarantee the correctness of the clipping. In fact, in absence of any priors on the weights, using local computations is not a viable option: for example, a seed with a very high weight can cover the entire domain.

Fortunately, in most applications (e.g., fluid dynamics), power

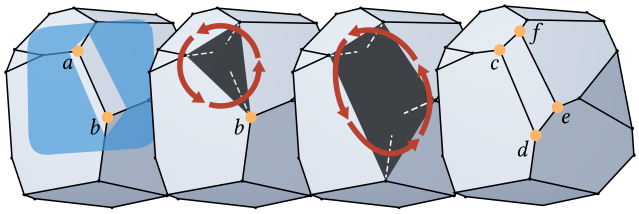

Figure 4: *Clipping a convex polyhedron (in gray) by a halfspace delimited by a plane (in blue). The clipping is done in three steps. First we detect the vertices to be removed (a and b). Then we iteratevely remove the vertices, thus creating a hole with dangling edges: we remove the vertex a and and initialize the hole boundary (red arrows) as the dual of the dangling edges. Then we remove the vertex b and update the boundary of the hole accordingly. Finally, we create the new facet and generate vertices for each dangling edge (in orange).*

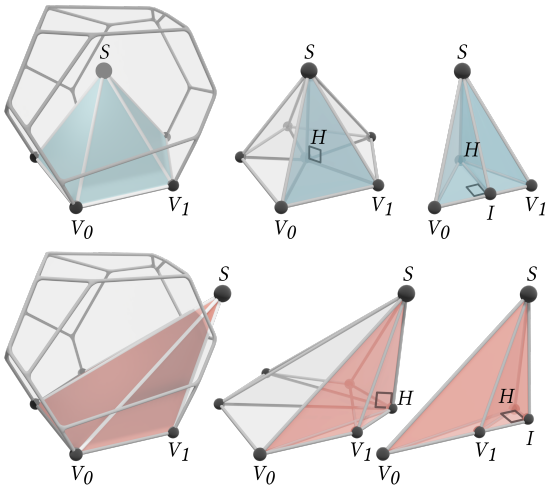

Figure 5: ***Top row:*** *to decompose the convex cell into tetrahedra, first we decompose it to pyramids associated with each Voronoi face (left image). Then we project the seed S onto the face (point H) and produce a tetrahedron per edge (middle image). These tetrahedra are split into two tetrahedra each by projecting S on the facet edge (point I, right image)* ***Bottom row:*** *if the seed S (or the projection H, or the projection I) is outside of the cell (resp. the face or the edge) some tetrahedra will have negative volume, thus balancing the extra volume from other tetrahedra. Note that seed S can be outside of the cell only with power diagrams.*

diagrams are often very similar to Voronoi diagrams, with this extra degree of freedom (the seed weights) that allows to translate the cells or adjust the volume of some cells. In both cases, the weights of neighboring seeds are very similar. To formalize this prior, we can assume that two seeds $\mathcal{S}_0, \mathcal{S}_1$ of weights w_0, w_1 are such that $\|w_0 - w_1\| < 2\varepsilon\|\mathcal{S}_0\mathcal{S}_1\|$. Under this assumption, adding ε to our security radius guarantees that the cell is completely computed.

Once the result is computed, we can check that ε was large enough by a simple volume comparison. If a cell was computed without taking into account a neighbor that should have participated, its volume will be too large. This allows to detect that ε is too low when the sum of the cell volumes is greater than the volume of Ω §6.3.

In our experiments on fluid simulation, setting ε to 0.2 times the security radius was sufficient to always succeed in computing all cells.

3. Integration over unrestricted cells

The previous section shows how to compute an unrestricted power diagram cell-by-cell. In this section, we recall how integrals Ray et al. evaluate the integrals. The principle is to compute the integral on each cell by a signed sum integrals over tetrahedra [LK84], where tetrahedra are obtained by orthogonal projections as in [MMdGD11].

Let us say we have a function $f(x) : \mathbb{R}^3 \to \mathbb{R}$ and for the cell $\mathcal{C}$ we want to compute the integral $\int_{\mathcal{C}} f\,dV$. The main idea is to decompose the polyhedron $\mathcal{C}$ into a set of tetrahedra $\mathcal{V}(\mathcal{C})$ and to evaluate the integral for each tetrahedron. The difficulty lies in the fact that the lightweight data structure used to represent the polyhedron does not allow for an efficient inference of connectivity between the edges and the faces. Fortunately, since we are not interested in the connectivity, but only in integrals over the cell, there is a way to compute them efficiently.

Assume that we split the polyhedron into pyramids having a cell face as the base and the seed $\mathcal{S}$ as the tip, as illustrated in Fig. 5. This pyramid can in turn be decomposed into a set of tetrahedra by creating a tet per edge V_iV_{i+1} of the base of the pyramid. In this case, the tetrahedron vertices are the edge vertices V_i and V_{i+1}, the seed $\mathcal{S}$ and its orthogonal projection H on the base of the pyramid.

With this approach, the problem of the connectivity still persists: we need to know that V_i, V_{i+1} is an edge. There is a way to circumvent the problem: if we introduce I, the orthogonal projection of the seed onto the edge V_iV_{i+1}; then the tetrahedron $(V_i, V_{i+1}, H, \mathcal{S})$ can be seen as a composition of two tetrahedra $(V_i, I, H, \mathcal{S})$ and $(I, V_{i+1}, H, \mathcal{S})$.

In this way, we can compute a decomposition $\mathcal{V}(\mathcal{C})$ without restoring the connectivity: for each vertex of the polyhedron we issue six tetrahedra. To compute the tetrahedra we only need to know the seed $\mathcal{S}$ and three plane equations incident to the vertex. This is not the minimal decomposition, but it can be generated directly from our lightweight data structure. Note that the orthogonal projections H and I can lie outside the pyramid's base and the corresponding edge, but signed volumes balance each other.

Thus, having the decomposition $\mathcal{V}(\mathcal{C})$, the integral can be computed as follows:

$$\int_{\mathcal{C}} f\,dV = \sum_{v \in \mathcal{V}(\mathcal{C})} \operatorname{sgn}(\operatorname{vol}(v)) \int_{v} f\,dV, \tag{1}$$

where $\operatorname{vol}(v)$ is the signed volume of the tetrahedron v. This is a

direct consequence of the divergence theorem applied to a radial vector field centered at $\mathcal{S}$. For example, if $f \equiv 1$, i.e. we want to compute the volume of the cell, we can take the field equal to $\vec{\mathcal{S}P}/6$ at point P. Indeed, the signed volume of v is equal to the flux of this field throw the triangle of v that is opposite to $\mathcal{S}$.

In Section 5, we detail the evaluation of the integrals needed for Lloyd's algorithm and fluid simulation: the cell's barycenter and volume and the area of the frontier between two cells.

4. Integrate over restricted cells

In the previous section we have shown how to integrate a function f over the intersection of a polyhedron $\mathcal{C}$ (the power diagram cell) and the bounding box of the pointset. Now, we are interested in more complex simulation domains: we want to replace the bounding box by a domain Ω represented by its triangulated boundary. The problem is to integrate the function f over the domain $\Omega \cap \mathcal{C}$.

We want to do this without changing our base framework, which computes the integral within regions defined as the intersections of half-spaces. The difficulty is how to define such half-spaces from an arbitrary enclosing surface mesh. To achieve this, we propose a decomposition exploiting signed integrals that combine and cancel out to provide the correct result.

The section is organized as follows: first we present in §4.1 the idea of our method without any optimization, and then in §4.2 we explain how to speed up the process by performing local computations only.

4.1. Without optimizations

Our goal is to compute the integral of a function f over the intersection $\mathcal{C} \cap \Omega$, i.e. $\int_{\Omega \cap \mathcal{C}} f\,dV$. We want to avoid the direct computation of $\Omega \cap \mathcal{C}$, so first let us study how to compute the integral $\int_{\Omega} f\,dV$.

The domain Ω is represented by a triangulated surface $\partial\Omega$; let us choose an arbitrary point O and define a set of tetrahedra $\mathcal{U}$ as follows: each triangle of $\partial\Omega$ creates a tetrahedron defined by three points of the triangle and the point O.

Using the same argument as for Eq. (1), we can write the integral as follows:

$$\int_{\Omega} f\,dV = \sum_{u \in \mathcal{U}} \mathrm{sgn}(\mathrm{vol}(u)) \int_{u} f\,dV.$$

This decomposition is illustrated in Fig. 6, where we compute the total volume of the bunny as the sum of volumes of blue tetrahedra ($\mathrm{sgn}(\mathrm{vol}(u)) > 0$) generated by triangles in $\partial\Omega_{in}$ minus the sum of the (unsigned) volumes of red tetrahedra ($\mathrm{sgn}(\mathrm{vol}(u)) < 0$) generated by $\partial\Omega_{out}$ (red).

This decomposition of Ω can also be used to integrate over the intersection $\mathcal{C} \cap \Omega$:

$$\int_{\mathcal{C}\cap\Omega} f\,dV = \sum_{u \in \mathcal{U}} \mathrm{sgn}(\mathrm{vol}(u)) \int_{\mathcal{C}\cap u} f\,dV.$$

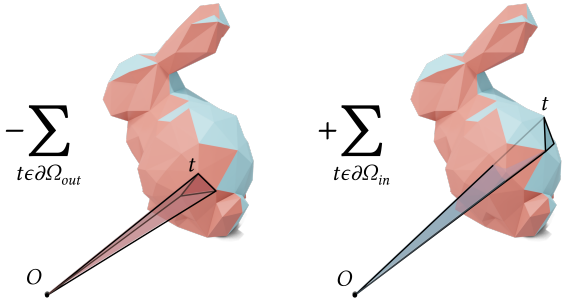

Figure 6: *An example of volume computation: when the normal of the triangle t points towards the point O (left), the volume of the tetrahedron is removed from the sum, otherwise (right) it is added.*

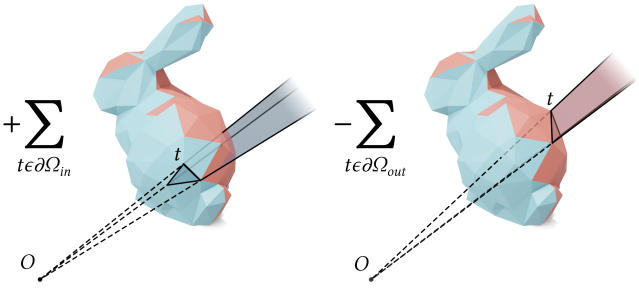

Figure 7: *The volume of the domain can also be computed as a sum of signed volumes of truncated cones generated by the triangles and the point O.*

Note that all the tetrahedra $u \in \mathcal{U}$ are convex polyhedrons, therefore, it is straightforward to compute the intersection $\mathcal{C} \cap u$: it simply means 4 halfspace intersections to add to the clipping procedure described in §2.2, and the integral itself can be computed with the Eq. (1):

$$\int_{\mathcal{C}\cap\Omega} f\,dV = \sum_{u \in \mathcal{U}} \mathrm{sgn}(\mathrm{vol}(u)) \sum_{v \in \mathcal{V}(\mathcal{C}\cap u)} \mathrm{sgn}(\mathrm{vol}(v)) \int_{v} f\,dV. \quad (2)$$

While the Eq. (2) allows us to compute the integral over the restricted convex cell $\Omega \cap \mathcal{C}$ with the aid of the set of tetrahedra $\mathcal{U}$, it is often computationally heavy. The simulation domain Ω is typically defined by thousands of triangles, making it impractical to intersect each power diagram cell with all the tetrahedra in $\mathcal{U}$. The main idea of the optimization is to early discard the empty cells $\mathcal{C} \cap u$ in the Eq. (2). The problem is that the set $\mathcal{U}$ does not allow us for a trivial detection.

To this end, we introduce a modified set $\mathcal{U}'$. If we consider the cone of tip $O \notin \Omega$ that would be split in two regions by the triangle $t \in \partial\Omega$: the tet $u \in \mathcal{U}$ was the first half that contains O, and now we consider the other (infinite) half $u \in \mathcal{U}'$.

It may be less intuitive to work with $\mathcal{U}'$ than with $\mathcal{U}$, but they share the key property: any point (in general position) outside the domain Ω belongs to an even number of polytopes (where integrals

cancel each other), whereas points inside the domain belong to an odd number of polytopes contributing to the integral.

Fig. 7 illustrates the volume computation ($f \equiv 1$) with the set $\mathcal{U}'$. The total volume of the bunny is equal to the signed sum of volumes of truncated cones u generated by the point O and the triangles in $\partial\Omega$.

Thus, we can plug $\mathcal{U}'$ instead of $\mathcal{U}$ in the Eq. (2):

$$\int_{\mathcal{C}\cap\Omega} f\,dV = \sum_{u\in\mathcal{U}'} \mathrm{sgn}(\mathrm{vol}(u)) \sum_{v\in\mathcal{V}(\mathcal{C}\cap u)} \mathrm{sgn}(\mathrm{vol}(v)) \int_v f\,dV. \quad (3)$$

Note that unbounded truncated cones $u \in \mathcal{U}'$ have infinite volume, however the sign is well-defined: for a truncated cone u generated by the triangle (t_0,t_1,t_2) and the point O we have $\mathrm{sgn}(\mathrm{vol}(u)) = -\mathrm{sgn}(\det 3x3(t_0-O,t_1-O,t_2-O))$. Also note that the Eq. (3) supposes $O \notin \Omega$; if this is not the case, we need to add the volume of the cell to the result, it is equivalent to creating an infinitesimal hole in Ω to exclude O from it.

From the computational point of view, the polytopes in $\mathcal{U}'$ are convex and bounded by 4 planes, thus it is straightforward to compute the intersection $u\cap\mathcal{C}$ for $u \in \mathcal{U}'$. If used as is, the Eq. (3) does not bring a direct gain over the Eq. (2), however it opens a window for an optimization.

4.2. Optimization for complex domains

The main goal of this section is to optimize the computations by doing them localy, i.e. using only the triangles situated near the cell $\mathcal{C}$. The key element to this optimization is to early discard the empty cells $\mathcal{C}\cap u$ in the Eq. (3).

To do so, we compute the bounding box $box(\mathcal{C})$ for each cell $\mathcal{C}$ (not necessarily a tight one). The point O is chosen to be the minimum coordinate corner of the box and thus the set $\mathcal{U}'$ is defined per cell. Note that any triangle $t \in \partial\Omega$ that does not intersect the box ($t\cap box(\mathcal{C}) = \emptyset$) produces a polytope u that does not intersect the cell ($u\cap\mathcal{C} = \emptyset$). Fig. 8 provides an illustration. Therefore, we can compute the Eq. (3) only for the polytopes generated by the triangles intersecting $box(\mathcal{C})$.

In practice, we pre-compute on the CPU the acceleration data-structure $dg \leftarrow domain\_grid(\partial\Omega)$ (Alg. 1, line 1). We place the domain Ω inside a regular grid of size N^3. The resolution is chosen to have only few triangles at most intersecting each voxel. For each voxel in the grid we pre-compute the list of the triangles in $\partial\Omega$ that intersect the voxel, as well as a boolean indicating whether the minimum coordinate corner of the voxel is inside Ω or not. We store the data in a compressed row storage matrix and in a vector of boolean values.

With this data structure, we compute integrals over restricted power cells with Alg. 2. First (Alg. 2, line 1), we choose the $box(\mathcal{C})$ to be the minimal bounding box of $\mathcal{C}$ composed of the voxels of dg. Next (Alg. 2, lines 2–3), we merge the list of the triangles intersecting $box(\mathcal{C})$ and we set the point O.

If O belongs to Ω, we add the integral over $\mathcal{C}$ to the result. Note that, in our scheduling strategy, the first substep of the first pass exits at this point and reports a failure if the list of triangles is not empty.

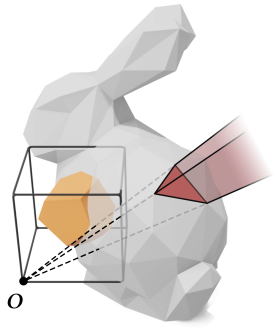

Figure 8: *When integrating over the orange polyhedron restricted to the bunny, any triangle (red) that is outside the black box (containing the polyhedron), generates a (red) polytope that does not intersect the orange polyhedron.*

The rest of the algorithm iterates over each triangle t, clips $\mathcal{C}$ by the 4 halfspaces that define the truncated cone created by the triangle t and O, and adds the corresponding integral if the normal of t points towards O, otherwise subtracts it.

Algorithm 2: Integrals over $\Omega\cap\mathcal{C}$

Input: dg; // *Domain grid*
Input: $\mathcal{C}$; // *Power diagram cell*
Output: I; // *Integrals*
1 $box(\mathcal{C}) \leftarrow dg.bounding\_subgrid(\mathcal{C})$;
2 $T \leftarrow \bigcup_{(i,j,k)\in box(\mathcal{C})} dg[i,j,k].triangles$;
3 $O \leftarrow \min_{(i,j,k)\in box(\mathcal{C})} (i,j,k)$;
4 $I \leftarrow 0$;
5 **if** $O \in \Omega$ **then** $I \leftarrow integrate(\mathcal{C})$;
6 **for** $t \in T$ **do**
7 $\quad \mathcal{C}_{clip} \leftarrow \mathcal{C}\cap truncated\_cone(O,t)$;
8 $\quad I \leftarrow I - \mathrm{sgn}(\det 3x3(t_0-O,t_1-O,t_2-O)) \times integrate\left(\mathcal{C}_{clip}\right)$;

The precomputation of the $domain\_grid(\Omega)$ is only reasonable if we need to compute multiple power diagrams restricted to a static domain, in this case it introduces a small overhead largely compensated by the gain during the integration phase.

Thus, Liu et al. evaluate integrals over each cell/tet intersection with a prunnng strategy to reduce unnecessary computations (at the expense of missing some intersections), whereas our algorithm needs to perform clipping for each boundary triangle intersecting a cell (typically zero or one per cell), thus offering a huge gain.

5. Applications

We show how to use our method in two applications: Lloyd's algorithm and incompressible fluid simulation. Each case requires to evaluate specific integrals.

5.1. Cell barycenter and volume

For Lloyd's algorithm, only the barycenter of each cell needs to be computed. It is equal to the integral of the coordinates x, y and z over the volume, divided by the volume of the cell. In practice, for each tetrahedron $v \in \mathcal{V}(\mathcal{C}_{clip})$, we compute its signed volume $\text{vol}(v)$ and barycenter $\text{bary}(v)$, and accumulate them on the fly. Then the barycenter of the polyhedron $\mathcal{C}_{clip}$ is given by:

$$\text{bary}(\mathcal{C}_{clip}) = \frac{\sum_{v \in \mathcal{V}(\mathcal{C}_{clip})} \text{vol}(v)\,\text{bary}(v)}{\sum_{v \in \mathcal{V}(\mathcal{C}_{clip})} \text{vol}(v)}.$$

The barycenters and the volumes of all polyhedra $\mathcal{C}_{clip}$ are, in their turn, accumulated on the fly to produce the barycenter and the volume of the cell $\mathcal{C}$ (Alg. 2, line 8).

5.2. Weighted Laplacian operator matrix

Power particles-like fluid simulations [dGWH*15b, GM17] enforce incompressibility of the fluid by computing a semi-discrete optimal transport. The integration over a power diagram is the bottleneck of the simulation.

In addition to the cell's barycenter and volume, semi-discrete optimal transport [dGBOD12, Mér11, Lév15, KMT16, LS18] requires evaluation of the weighted Laplacian operator. To evaluate this matrix, for every pair of adjacent cells, we need the area of the frontier, its barycenter and the distance between the corresponding seeds.

We can represent this information by a sparse matrix, providing all necessary values for each pair of seeds. In order to compute this matrix, we keep track of the neighboring seeds every time we generate a halfspace equation. This requires us to maintain an array of seeds synchronized with the array of halfspace equations. Having the array, once the cell is computed, the distance from the seed to its neighbors is straightforward to compute. Just as before, the decomposition of the cells into tetrahedra allows to compute the area of the frontiers as well as the corresponding barycenters. In practice, the symmetry of the matrix allows to compute only the upper part of the matrix (when the neighbor seed id is larger than the current seed id). The output matrix is stored by a fixed size array of non null coefficient for each seed.

6. Experiments and discussion

The efficiency of our algorithm strongly relies on the regularity of the distribution of seeds and weights. This section is organized as follows. First we explain how we have tuned the algorithm parameters (§6.1). Then we evaluate the performance of our algorithm (§6.2) in different settings, by varying the integration domain, the type of seed distribution and Voronoi diagram / power diagram. Finally, we conclude by presenting the limitations and some possible future improvements (§6.3).

6.1. Parameters tuning

To run the algorithm, we need to choose three parameters; for maximum performance we vary them as a function of priors on data we have:

- The parameter K is the number of neighbors that are computed for each seed. If K is large, the running time and the memory consumption increase, but if there are no enough neighbors to reach the security radius, the algorithm fails to compute the cell.
- The parameter P is the maximum number of halfspaces that intersect a cell during its construction. It affects the memory allowed per thread and therefore the maximum number of threads that can access simultaneously to shared memory.
- The parameter V is the maximum number of vertices of the cell. It also impacts the memory usage per thread, with the same consequences on performances.

The best situation for our algorithm would be a blue noise with a constant weight per seed (i.e. a Voronoi diagram), but in practice inputs are not always that convenient. For Lloyd's algorithm, the seed distribution in the first iterations may be a white noise. For fluid simulations, the noise is closer to blue, but the difference of neighbor seed weights requires us to consider larger neighborhoods.

For each situation, the parameters K, P, V should be tuned for optimal performances. Their values must be large enough to fail only on a small percentage of Voronoi cells, and as low as possible to minimize the time to query all neighbors (with parameter K) and the memory consumption (for all parameters).

We found our settings by an experimental approach: we run our algorithm with values of K, P, V that compute all cells without failure and register the required value of K, P, V for each cell. It gave us a distribution of K, P and V for each configuration and we set K, P, V to succeed in $\approx 95\%$ of the cells. In practice, K varies a lot according to the type of seed and weight distribution, but P and V are just a bit lower for the blue noise.

Our test cases are using the 3 seed distributions introduced in [RSLL18] (blue noise, white noise and something in-between obtained by a perturbed grid) with 10M seeds. Fig. 9 provides the histograms of K, P, V for each distribution for both Voronoi diagrams and power diagrams. The weights of the power diagrams were computed to produce the equal volume for all the cells in the diagram.

6.2. Timings

This work extends the state of the art of Voronoi Diagrams computation on the GPU [RSLL18] that assumes a reasonable seed distribution.

Our OpenCL implementation uses 64-bit floating-point numbers. In this article we present experiments performed on an Nvidia Tesla V100 ; note that it can be perfectly done on a consumer range GPU (ex. GTX1080) which does not have rapid double type. On a GTX1080 our choice of 64-bit floating-point numbers roughly doubles the running time w.r.t 32-bit floats, but it reduces drastically (virtually eliminates) the number of CPU fallbacks (when in-sphere predicate fails).

The results are summarized in Fig. 10. We have perfomed the computations on three datasets (10M seeds) with different types of point distributions (blue noise, perturbed grid, white noise). The

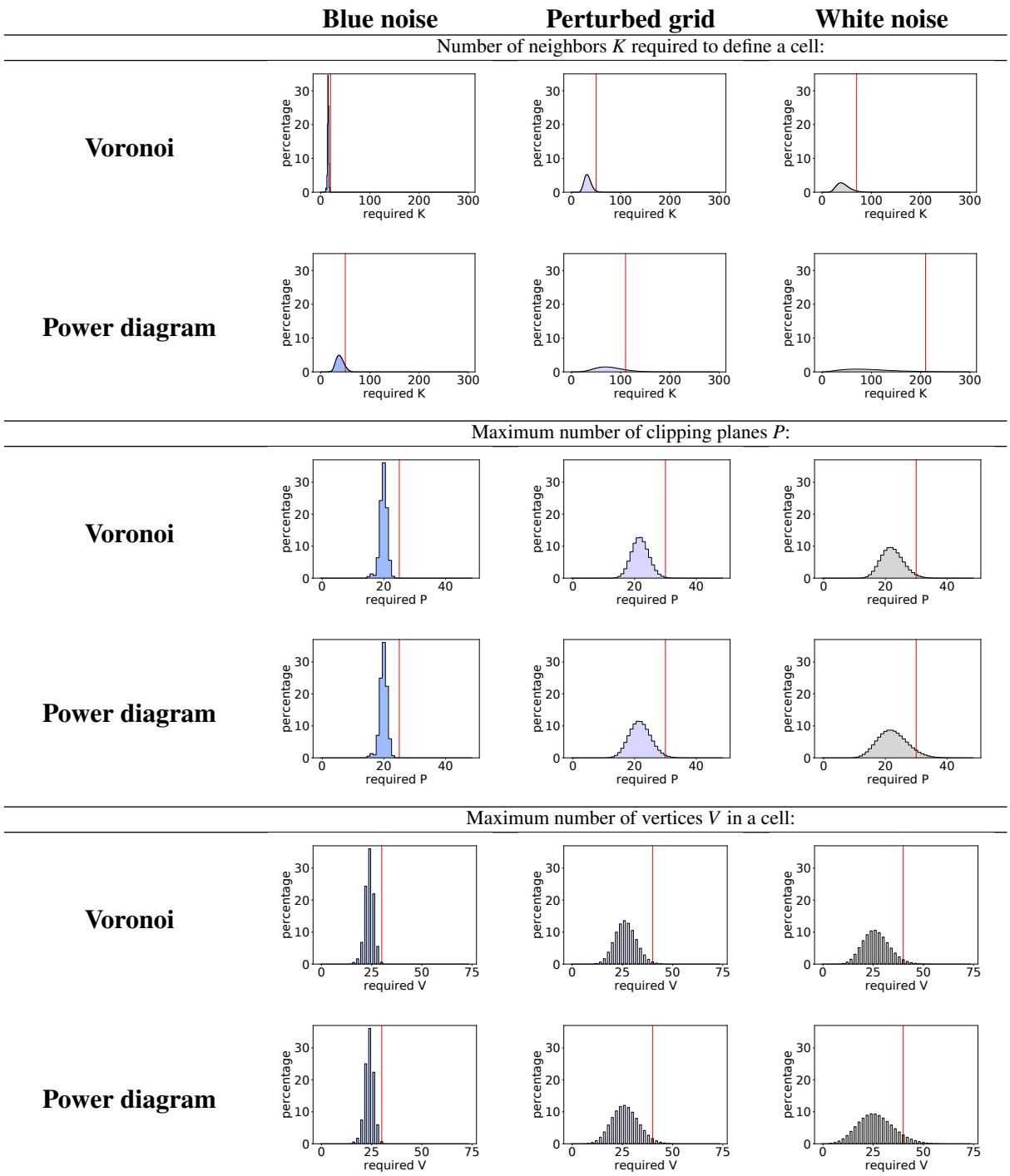

Figure 9: *Statistics on 10M Voronoi and 10M power diagram cells.* ***The columns*** *correspond to three datasets generated with the blue noise, perturbed grid and the white noise.* ***Top:*** *the histograms of the distribution of the number of neighbors before reaching the security radius criteria.* ***Middle:*** *maximum number of planes during the clipping of a cell.* ***Bottom:*** *maximum number of vertices during the clipping. Red lines show the parameter values we have chosen for the first pass of the algorithm.*

leftmost column provides the performances of [RSLL18] (with 32-bit floating-point); the second column gives the running times of our algorithm without the new features: we have computed Voronoi diagrams without domain restrictions. Our scheduling strategy allows to use better parameters for most cells, resulting in a speedup factor of 2 to 3 according to the type of seed distribution.

The 6 rightmost columns correspond to the Voronoi diagrams restricted to different domains. We observe that adding the restriction introduces a reasonable overhead (typically inferior to 15%). Note that the first pass is performed in two substeps: in the first substep we compute only the cells that are guaranteed to lie entirely inside or outside the domain and the second substep computes the rest of the cells. The bar chart shows the first substep in solid blue and the second substep in hatched blue.

For computing power diagrams we need to increase the security radius, in our experiments we increased it by a factor 1.2 for fluid simulations [dGWH*15a, GM17]. It may seem like a minor difference, but it basically scales the number of neighbors to consider by 1.2^3 which almost doubles the region where neighbor seeds can participate to a power cell. We observe (Fig. 11) an impact on performances that is proportional to the neighborhood size increment: almost a factor of two. For higher variation of weights, a more complex strategy would be required to avoid crushing the performances.

6.3. Discussion

Compared to other GPU algorithms. For computing unrestricted Voronoi diagrams, our algorithm provides a speed up factor of two over [RSLL18] in the worst case. The recent GPU implementation of restricted Voronoi diagrams [LMGY20] processes 60K seeds per second (30K seeds in 0.52 seconds) in the most favorable case. The fact that we work directly with the boundary of the domain allows us to process 4300K seeds per second (10M seeds in 2.31 seconds) in the worst case, with a similar domain restriction. Moreover, our algorithm does not suffer from missing Voronoi cell-tet pairs to intersect, thus providing accurate results.

In addition to that, we provide all the features needed for computing a semi-discrete optimal transport in a domain (power cells, restriction to Ω and new integrals).

Compared to CPU algorithms. Our solution is an order of magnitude faster for our use cases. However, we exploit priors on seed distribution and power diagram weights. For a general purpose usage, CPU implementations remain a better option.

On a 3.40GHz, 12 cores Intel i7-6800K CPU a multithreaded Geogram implementation takes 41.9 seconds to compute the power diagram from the Fig. 11, right (against 0.79 seconds on the Tesla V100). 26.6 seconds were spent computing the Laguerre diagram and 15.3 seconds to clip it. CGAL timings are similar.

Predicate robustness. This is a common issue in power diagram algorithms. As in [RSLL18], a CPU fallback is always possible. In practice, we observed numerical failures only with 32-bit floating-point and upgrading to 64-bit solved all our problems.

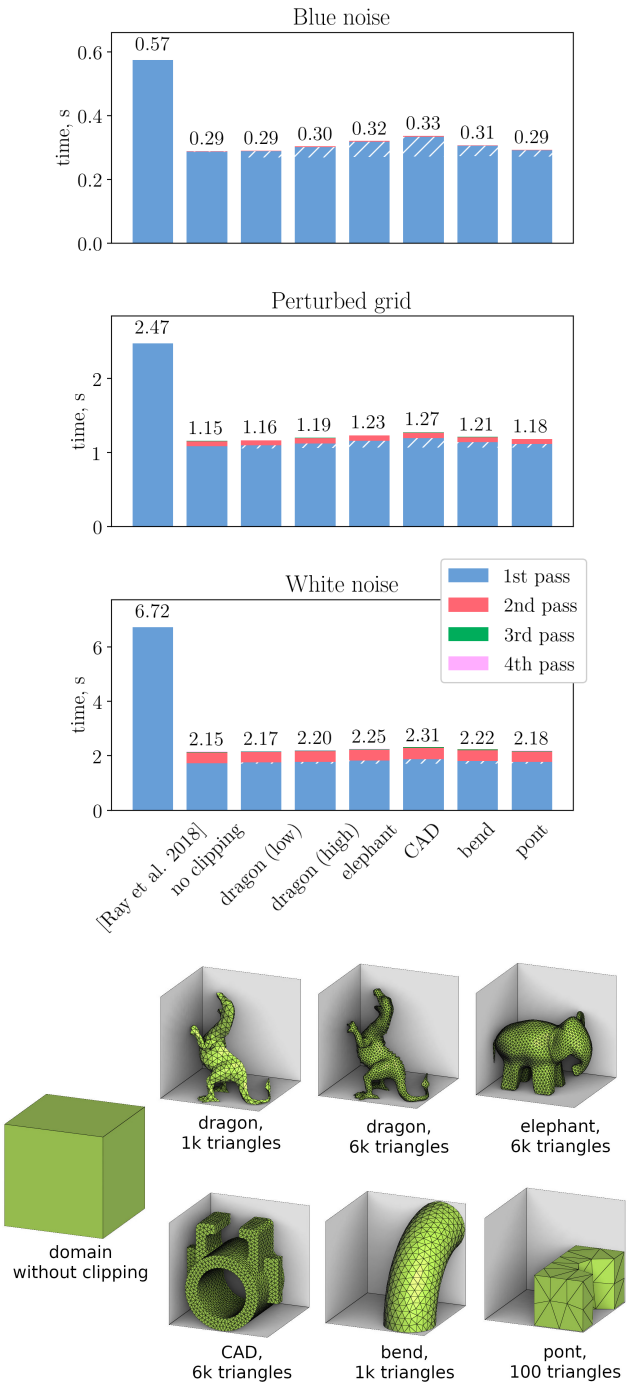

Figure 10: *Statistics on 10M Voronoi seeds with blue noise distribution **(top)**, located on a perturbed grid **(middle)** and white noise **(bottom)**. **The leftmost column** corresponds to the running time of the algorithm [RSLL18]; **the second column** is our result without clipping; **6 rightmost columns** correspond to different clipping domains.*

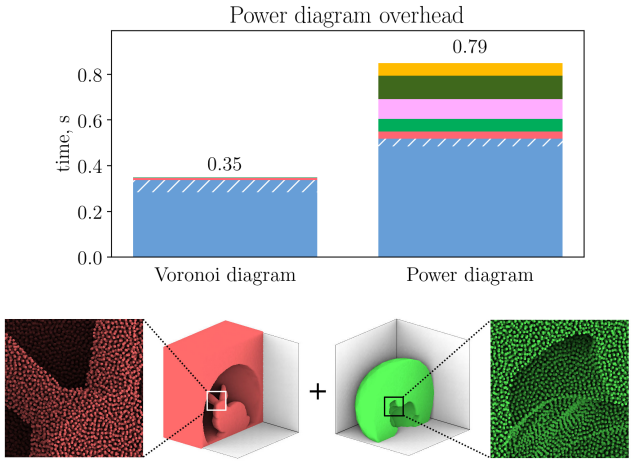

Figure 11: *Overhead necessary for computing a power diagram.* ***Left:*** *Voronoi diagram computed on 10M seeds (blue noise), restricted to the bunny snowglobe. This diagram has ≈ 4M non-empty Voronoi cells, the seeds are shown in green in the bottom image. The seeds producing empty cells are shown in red.* ***Right:*** *Power diagram, the weights of the 4M seeds are computed to produce an equal volume for all cells.*

Floating-point integration precision. It might seem that our algorithm fundamentally suffers from what is called "catastrophic cancellation" arising from the sum and subtraction from partitioned volume elements. Indeed, if the point O (refer to Fig. 8) was badly chosen (e.g. the origin), it might be the case. For each cell, we choose the point as the minimum coordinate corner of the bounding box of the cell, thus the point it is not far from the cell, greatly improving the precision.

To validate the precision of our results, we have performed a test where we compare our results with the results computed on a CPU. We have computed the volume and the barycenter for each power diagram cell for the dataset shown in Fig. 11. This computation was made both on the CPU (with a Geogram implementation) and on the GPU (with our method) with 64-bit floating point numbers.

So, we have two sets of `double` values $\left\{v_i^{CPU}\right\}_{i=1}^{n}$ and $\left\{v_i^{GPU}\right\}_{i=1}^{n}$, that correspond to the volume of non-empty cells computed on the CPU and the GPU respectively, where n stands for the number of non-empty cells. We also have two sets of `double3` values $\left\{b_i^{CPU}\right\}_{i=1}^{n}$ and $\left\{b_i^{GPU}\right\}_{i=1}^{n}$ corresponding to the cell barycenters.

In addition to that, we have computed on the CPU the volume of the domain V (`double`) and its barycenter B (`double3`). This computation was made directly over the domain. First we recompute the invariants from the GPU data and we compare the precision:

$$\frac{\left|\sum_i v_i^{GPU} - V\right|}{V} \approx 3 \cdot 10^{-15}$$

Then for each coordinate c we check the difference between both ways to compute the barycenter of the domain:

$$\max_{c \in x,y,z} \frac{\left|\sum_i \frac{v_i^{GPU} b_i^{GPU}[c]}{V} - B[c]\right|}{B[c]} \approx 3 \cdot 10^{-14}$$

Next we compare two power diagrams. First we compare extreme values of discordance:

$$\frac{\max_i \left|v_i^{GPU} - v_i^{CPU}\right|}{V/n} \approx 6 \cdot 10^{-11}$$

$$\max_{c \in x,y,z} \max_i \frac{\left|b_i^{GPU}[c] - b_i^{CPU}[c]\right|}{B[c]} \approx 2 \cdot 10^{-6}$$

We terminate the evaluation by computing the average discordance:

$$\frac{\sum_i \left|v_i^{GPU} - v_i^{CPU}\right|}{V} \approx 3 \cdot 10^{-12}$$

$$\max_{c \in x,y,z} \sum_i \frac{\left|b_i^{GPU}[c] - b_i^{CPU}[c]\right|}{nB[c]} \approx 2 \cdot 10^{-11}$$

Thus, our GPU algorithm offers a fair precision in numerical computation of integrals.

Prior on power diagram weights. To compute power diagram cells, we can ensure that enough neighbors are visited by simply increasing the security radius criteria used for Voronoi cells. This strategy has proven to be efficient for our tests on fluid simulations, but it relies on a strong assumption on bounds of weights variations. Other applications may need heuristics to locally adapt this security radius, or use other strategies according to their own variation of weights.

Note that there is a way to assert the correctness of the power diagram computation. If the security radius criterion does not reflect the variation of weights, it means that some halfspace intersections will be missed. In this case the sum of volumes of the power diagram cells will be greater than the volume of the domain. It is possible to narrow down the problematic zone and increase the security radius locally.

Conclusion

Being able to optimize a mesh by numerical methods opens new perspective to scale up key applications. We have explored the effectiveness of our technique in sampling problems and for solving semi discrete optimal transport. This work makes it possible to evaluate integrals over the volume enclosed by a mesh much faster than equivalent CPU solutions and requires less priors on seed and weight distributions than previous works on GPU.

The source code is available on github: basselin7u/GPU-Restricted-Power-Diagrams.

References

[ACR03] AMENTA N., CHOI S., ROTE G.: Incremental constructions con BRIO. In *Proceedings of the 19th ACM Symposium on Computa-*

tional Geometry, San Diego, CA, USA, June 8-10, 2003 (2003), pp. 211–219. URL: http://doi.acm.org/10.1145/777792.777824, doi:10.1145/777792.777824.

[AR95] ALEXANDER S., RAINALD L.: Three-dimensional parallel unstructured grid generation. *International Journal for Numerical Methods in Engineering 38*, 6 (1995), 905–925.

[Aur87] AURENHAMMER F.: Power diagrams: Properties, algorithms and applications. *SIAM Journal on Computing 16*, 1 (1987), 78–96. doi:10.1137/0216006.

[BMPS10] BATISTA V. H., MILLMAN D. L., PION S., SINGLER J.: Parallel geometric algorithms for multi-core computers. *International Journal for Numerical Methods in Engineering 43*, 8 (2010), 663–677.

[Bow81] BOWYER A.: Computing dirichlet tessellations. *Comput. J. 24*, 2 (1981), 162–166. URL: http://dx.doi.org/10.1093/comjnl/24.2.162, doi:10.1093/comjnl/24.2.162.

[Cao14] CAO T.-T.: Fundamental computational geometry on the gpu, 2014.

[CNGT14] CAO T.-T., NANJAPPA A., GAO M., TAN T. S.: 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* (2014), pp. 47–54. URL: http://doi.acm.org/10.1145/2556700.2556710, doi:10.1145/2556700.2556710.

[DCS99] DE COUGNY H. L., SHEPHARD M. S.: Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering 46*, 7 (1999), 1101–1125.

[DD18] DELAGE C., DEVILLERS O.: Spatial sorting. In *CGAL User and Reference Manual*, 4.12.1 ed. CGAL Editorial Board, 2018. URL: https://doc.cgal.org/4.12.1/Manual/packages.html#PkgSpatialSortingSummary.

[dGBOD12] DE GOES F., BREEDEN K., OSTROMOUKHOV V., DESBRUN M.: Blue noise through optimal transport. *ACM Trans. Graph. 31*, 6 (Nov. 2012). URL: https://doi.org/10.1145/2366145.2366190, doi:10.1145/2366145.2366190.

[dGWH*15a] DE GOES F., WALLEZ C., HUANG J., PAVLOV D., DESBRUN M.: Power particles: an incompressible fluid solver based on power diagrams. *ACM Trans. Graph. 34*, 4 (2015), 50:1–50:11. URL: http://doi.acm.org/10.1145/2766901, doi:10.1145/2766901.

[dGWH*15b] DE GOES F., WALLEZ C., HUANG J., PAVLOV D., DESBRUN M.: Power particles: An incompressible fluid solver based on power diagrams. *ACM Trans. Graph. 34*, 4 (July 2015), 50:1–50:11. URL: http://doi.acm.org/10.1145/2766901, doi:10.1145/2766901.

[FRWW14] FEI Y., RONG G., WANG B., WANG W.: Parallel l-bfgs-b algorithm on gpu. *Computers & Graphics 40* (2014), 1 – 9. URL: http://www.sciencedirect.com/science/article/pii/S0097849314000119, doi:https://doi.org/10.1016/j.cag.2014.01.002.

[GM17] GALLOUËT T. O., MÉRIGOT Q.: A Lagrangian scheme à la Brenier for the incompressible euler equations. *Foundations of Computational Mathematics* (May 2017). URL: https://doi.org/10.1007/s10208-017-9355-y, doi:10.1007/s10208-017-9355-y.

[Gon16] GONZALEZ R. E.: Paravt: Parallel voronoi tessellation code. *Astronomy and Computing 17* (2016), 80–85. URL: http://www.sciencedirect.com/science/article/pii/S2213133716300609, doi:https://doi.org/10.1016/j.ascom.2016.06.003.

[Inr18] INRIA P. A.-P.: Geogram: a programming library of geometric algorithms. http://alice.loria.fr/software/geogram/doc/html/index.html, 2018.

[KMT16] KITAGAWA J., MÉRIGOT Q., THIBERT B.: A newton algorithm for semi-discrete optimal transport. *CoRR abs/1603.05579* (2016). URL: http://arxiv.org/abs/1603.05579, arXiv:1603.05579.

[LB13] LÉVY B., BONNEEL N.: Variational anisotropic surface meshing with voronoi parallel linear enumeration. In *Proceedings of the 21st International Meshing Roundtable* (Berlin, Heidelberg, 2013), Jiao X., Weill J.-C., (Eds.), Springer Berlin Heidelberg, pp. 349–366.

[Lév15] LÉVY B.: A numerical algorithm for l2 semi-discrete optimal transport in 3d. *ESAIM: Mathematical Modelling and Numerical Analysis 49*, 6 (2015), 1693–1715.

[LK84] LIEN S., KAJIYA J. T.: A symbolic method for calculating the integral properties of arbitrary nonconvex polyhedra. *IEEE Computer Graphics and Applications 4*, 10 (1984), 35–42.

[LMGY20] LIU X., MA L., GUO J., YAN D.-M.: Parallel computation of 3d clipped voronoi diagrams. *IEEE Transactions on Visualization and Computer Graphics PP* (07 2020), 1–1. doi:10.1109/TVCG.2020.3012288.

[LS18] LÉVY B., SCHWINDT E. L.: Notions of optimal transport theory and how to implement them on a computer. *Computers & Graphics 72* (2018), 135–148. URL: https://doi.org/10.1016/j.cag.2018.01.009, doi:10.1016/j.cag.2018.01.009.

[LY19] LIU X., YAN D.-M.: Computing 3d clipped voronoi diagrams on gpu. In *SIGGRAPH Asia 2019 Posters* (New York, NY, USA, 2019), SA '19, ACM, pp. 9:1–9:2. URL: http://doi.acm.org/10.1145/3355056.3364581, doi:10.1145/3355056.3364581.

[MDL16] MARTÍNEZ J., DUMAS J., LEFEBVRE S.: Procedural Voronoi Foams for Additive Manufacturing. *ACM Transactions on Graphics 35* (2016), 1 – 12. URL: https://hal.archives-ouvertes.fr/hal-01393741, doi:10.1145/2897824.2925922.

[Mér11] MÉRIGOT Q.: A multiscale approach to optimal transport. *Comput. Graph. Forum 30*, 5 (2011), 1583–1592.

[MMdGD11] MULLEN P., MEMARI P., DE GOES F., DESBRUN M.: Hot: Hodge-optimized triangulations. In *ACM SIGGRAPH 2011 Papers* (New York, NY, USA, 2011), SIGGRAPH '11, Association for Computing Machinery. URL: https://doi.org/10.1145/1964921.1964998, doi:10.1145/1964921.1964998.

[MPR18] MAROT C., PELLERIN J., REMACLE J.-F.: One machine, one minute, three billion tetrahedra. https://arxiv.org/abs/1805.08831, 2018.

[ND03] NIKOS C., DAMIAN N.: Parallel delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering 58*, 2 (2003), 161–176.

[Rem17] REMACLE J.-F.: A two-level multithreaded delaunay kernel. *Computer-Aided Design*, 85 (2017), 2–9.

[RSLL18] RAY N., SOKOLOV D., LEFEBVRE S., LÉVY B.: Meshless voronoi on the gpu. In *SIGGRAPH Asia 2018 Technical Papers* (New York, NY, USA, 2018), SIGGRAPH Asia '18, ACM, pp. 265:1–265:12. URL: http://doi.acm.org/10.1145/3272127.3275092, doi:10.1145/3272127.3275092.

[Ryc09] RYCROFT C.: Voro++: A three-dimensional voronoi cell library in c++. 041111.

[The18] THE CGAL PROJECT: *CGAL User and Reference Manual*, 4.12.1 ed. CGAL Editorial Board, 2018. URL: https://doc.cgal.org/4.12.1/Manual/packages.html.

[Wan17] WANG L.: *Algorithms and Criteria for Volumetric Centroidal Voronoi Tessellations*. PhD thesis, 01 2017.

[Wat81] WATSON D.: Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *Comput. J. 24*, 2 (1981), 167–172.

[XLC*16] XIN S.-Q., LÉVY B., CHEN Z., CHU L., YU Y., TU C., WANG W.: Centroidal power diagrams with capacity constraints: computation, applications, and extension. *ACM Trans. Graph. 35*, 6 (2016), 244:1–244:12. URL: http://dl.acm.org/citation.cfm?id=2982428.