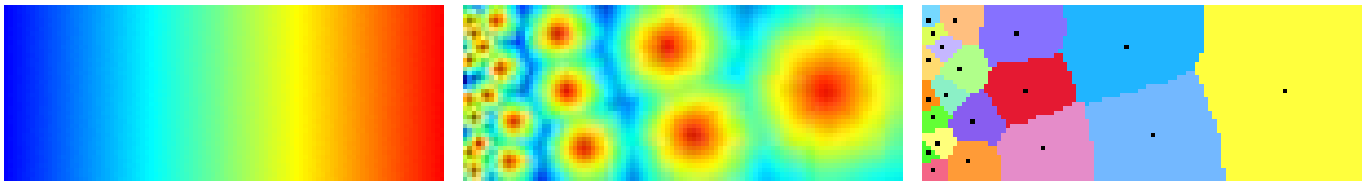
Parallel Shortest Path Algorithm for Voronoi Diagrams with Generalized Distance Functions

Julio Toss, João Luiz Dihl Comba, Bruno Raffin

Parallel Shortest Path Algorithm for Voronoi Diagrams with Generalized Distance Functions

Julio Toss, João Comba
Institute of Informatics
Federal University of Rio Grande do Sul, UFRGS
Porto Alegre - RS, Brazil
Email: {jtoss,comba}@inf.ufrgs.br

Bruno Raffin
Inria, CNRS
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
Email: Bruno.Raffin@inria.fr



(a) Stiffness gradient from soft (blue) to stiff (red) (b) Compliance distance: far (blue) to close (red) (c) Voronoi diagram

Fig. 1. Computing the Voronoi diagram with generalized distance functions on a voxel grid with dimensions 100x40x20 for 20 seeds: The distance function is defined on a non-Euclidean space which depends on specific material properties, in this example a gradient that encodes material stiffness (a). The resulting distance map (b) is obtained after computing the shortest path on the voxel grid from each Voronoi seed (c). The GPU grid-based parallel shortest path algorithm we propose was 27 times faster than the reference sequential CPU implementation.

***Abstract*—Voronoi diagrams are fundamental data structures in computational geometry with applications on different areas. Recent soft object simulation algorithms for real time physics engines require the computation of Voronoi diagrams over 3D images with non-Euclidean distances. In this case, the computation must be performed over a graph, where the edges encode the required distance information. But excessive computation time of Voronoi diagrams prevent more sophisticated deformations that require interactive topological changes, such as cutting or stitching used in virtual surgery simulations. The major bottleneck in the Voronoi computation in this case is a shortest-path algorithm that must be computed multiple times during the deformation.**

In this paper, we tackle this problem by proposing a GPU algorithm of the shortest-path algorithm from multiple sources using generalized distance functions. Our algorithm was designed to leverage the grid-based nature of the underlying graph used in the simulation. Experimental results report speed-ups up to 65x over a current reference sequential method.

***Keywords*-Voronoi Diagram; GPGPU; Parallel Programming; Physics Based Simulation.**

I. Introduction

The Voronoi diagram is a classical subdivision of space that is suitable for answering proximity problems, such as finding the nearest site, facility location, motion planning, coverage in sensor networks, etc. There are many variations of Voronoi diagrams, which are often associated to the underlying distance measure used. In this work we are particularly interested in the computation of Voronoi diagrams for non-Euclidean spaces, where a generalized distance function is used. This motivation came from the area of physics-based simulation, where Voronoi diagrams were used by Faure et at. [1] to simulate meshless deformable objects with heterogeneous material properties and complex geometries. Their proposal relies on a novel method which uses material-aware shape functions to describe the composition of simulated bodies, which can be composed of both soft and stiff materials.

The computation of Voronoi diagrams plays a central role in their proposal. To have accurate and realistic deformations, the underlying deformation space is discretized into a grid, and the material stiffness is defined for each vertex of the grid (material map). The deformation algorithm considers grid vertices as simulation nodes, with an associated Voronoi kernel function that limits the region of influence of the node. Since the distance function is not computed in a standard Euclidean plane, it must be scaled according to compliance values in the material map. Therefore, points connected by similar materials (i.e. inside the same Voronoi cell) will deform in a similar way.

The computation of the Voronoi kernel function and hence the Voronoi diagram is done during the setup phase of the simulation and remains unchanged during the whole simulation, as long as the topology of the object and material properties do not change. This allows the subsequent simulation phase to be performed in real-time, a necessary requirement for interactive applications. However, to enable interactive changes in the topology of the object (e.g. cutting or stitching), the Voronoi diagram must be recomputed during the simulation. This is not possible in their solution, since they report initialization times ranging from less that 1 second for a grid of 100x40 voxels to 10 minutes for a 500x200 grid. However, their implementation

is strictly sequential leaving plenty of room for optimization and parallelization.

Contributions: In this work we explore certain properties of the deformation problem to speed up the computation of shortest paths in graphs, which are used in the Voronoi diagram computation. Our proposal describes a shortest path algorithm using a parallel implementation that leverages the processing power of Graphics Processing Units (GPUs). Current GPU proposals for shortest path algorithms consider the Single-Source-Shortest-Path (SSSP) problem [2], using classic algorithms such as Dijkstra [3] or Bellmann-Ford. Unlike these algorithms, our algorithm considers the Multiple-Source-Shortest-Path (MSSP), since multiple shortest path computations are triggered at each simulation node. Moreover, we leverage the fact that our shortest-path algorithm can be computed over a grid instead of general graphs (used in current algorithms), which allows the algorithm to be more efficient.

We implemented our algorithm in CUDA and tested different design decisions with a collection of simulation examples. We prepared an experimental evaluation comparing our GPU implementation against the sequential CPU reference method, and obtained speed-ups ranging form 3x for small inputs up to 65x for larger ones, both with synthetic and real datasets.

II. Related work

The Voronoi diagram is a data structure extensively studied in the context of computational geometry for many different applications. Originally it defines a region of proximity for a set of k sites (seeds) in a plane where the distance between points is defined by their Euclidean distance. Most works use methods to efficiently compute them on a contiguous Euclidean space [4], or to compute their discrete approximation [5], [6].

Although most works in the literature compute Voronoi diagrams on Euclidean spaces, there are generalizations in the context of graphs [7], [8], sometimes called the *Graph Voronoi Diagram* [7]. In this context, the distance metric considered corresponds to the shortest path between nodes. This formulation of Voronoi diagrams often arises in the field of Facility Location, where clients and suppliers lie in an interconnection network. Computing these Voronoi diagrams basically consists in concurrently computing shortest paths from multiple sources on a weighted graph.

A. Parallel Voronoi diagrams computation

Over the last decade, the advent of parallel processors motivated the scientific community to put its effort in creating parallel algorithms for classical problems. At the same time, GPUs evolved to become general purpose massive parallel processors attracting attention from other fields of computing. Early study made by [5] used graphics hardware (pre-CUDA) to compute an approximation of the Voronoi diagram. Most works found in the literature rely on a discrete approximation, usually either on 2D pixel-maps or on surfaces.

The many parallel approaches vary in the way the information of proximity from the Voronoi centers is propagated to each pixel. *Jump Flooding Algorithm* (JFA) is proposed by [4], [6] as an algorithmic paradigm for GPGPU with application on Voronoi diagram computation. In JFA, the seeds start propagating their coordinates to neighbor pixels according to a pattern that halves the offset at each step. Each pixel compares the new information received with the current one and keeps the coordinates of the closest seed. In this case, distances can easily be computed on the Euclidean plane.

Weber et al. [9] introduces an interesting parallel algorithm called *parallel marching method (PMM)* to compute distances on surfaces with application on Voronoi diagrams. This method is indeed an extension of the fast marching method which is based on a priority-queue. However, this kind of data structure is difficult to efficiently parallelize. Instead, their method uses a specific traversing order of the grid, called *raster scan*, which shows an efficient parallelization algorithm.

More recently, [10] proposed a substantially different method from the previous ones. It uses a combinatorial approach to compute, in parallel, the exact polygons that form each cell of the Voronoi diagram. This work however is mainly theoretical, showing formal proofs without bringing experimental results.

All of these works consider distance computation on the Euclidean space, none of them deals with Voronoi diagrams on the graph space. This means that the shortest distances are always straight lines, hence these methods cannot be directly applied on graph problems. To the best of our knowledge there are no algorithm addressing the parallelization of the *Graph Voronoi Diagram*.

B. Parallel graph algorithms

Distance computation for the construction of the Voronoi diagram defined over graphs require finding shortest paths (with multiple sources plus concurrent search) [7]. The original Dijkstra's sequential algorithm for solving the Single-Source Shortest-Path (SSSP) [3] has $O(V^2)$ complexity on the number of vertices, while the min-priority-queue based version has complexity $O(E+VlogV)$, where E is the number of edges.

Several parallel approaches to solve the SSSP problem have been proposed on the literature. Crauser et Al. [11] proposed a parallel PRAM algorithm of Dijkstra's which needs $O(n^{1/3}log\ n)$ time. However, Dijkstra is an inherently sequential algorithm, with lots of synchronizations with no efficient PRAM implementation [2]. Alternatively, other works parallelize the SSSP based on Bellman-Ford's algorithm which is less efficient than Dijkstra on sequential implementations, but has a higher degree of parallelism [12], [13].

Most of the existing parallel SSSP algorithms have to deal with a trade-off between the amount of parallelism exposed and the extra work generated. The parallel $delta-stepping$ method, proposed on [14], has a good compromise between these two factors. They report an implementation exhibiting 30x speed-up on a CRAY MTA-2 shared memory architecture with 40 processors.

1: **procedure** RELAX(u, v, w)
2: **if** $v.d > u.d + w(u, v)$ **then**
3: $v.d \leftarrow u.d + w(u, v)$
4: **end if**
5: **end procedure**

Fig. 2. Relaxation Algorithm

C. GPU implementation of SSSP algorithms

Several GPU implementations have been proposed over the last years for different graph algorithms [2]. For the shortest path problem specifically, Dijkstra-based parallelizations are more frequently used [2], [14], [15]. Although, other approaches exist (e.g [12]), which proposes a modified Bellman-Ford algorithm on GPU for dense graphs.

In general, Dijkstra based algorithms use a technique known as edge relaxation. In this technique each vertex maintains an estimate shortest-path with distance $v.d$. The process of relaxation consist of trying to improve this estimate by going from vertex u to v through an edge of weight $w(u, v)$ (Fig. 2).

When done in parallel, each vertex u is assigned to a thread which may update $v.d$ concurrently, thus creating a critical section. Consequently, lines 2 - 4, of algorithm in Fig. 2, have to be protected in an atomic region. In modern CUDA devices this atomic region can be efficiently implemented by the single atomic instruction $atomic\_min(addr, val)$ [1] .

III. Parallel Graph Voronoi algorithm

As mentioned on the previous section, the Graph Voronoi can be seen as an extension of the shortest path problem. However its parallelization poses additional problems of concurrent access on shared variables. In the Voronoi diagram problem, each voxel has to keep the distance estimate value to the seed and an extra variable for its Voronoi cell index. These variables would then be updated serially in the relaxation procedure, which, if executed by two threads in parallel, could lead to any combination of results in these variables. On concurrent programming this is a classical case of race condition. The straightforward solution for this problem would be to enclose the whole critical section (Fig. 2) within mutex locks. However, mutexes are expensive structures to implement on GPUs. To deal with this problem, we choose to encode both variables, Voronoi index and distance estimate, in a single 32-bit word (Fig. 3) which can then be atomically updated in a single *atomic_min()* instruction. Our encoding can be adjusted to balance distance precision and maximum number of Voronoi cells. In our implementation, we reserved 24 bits for the distance and 8 bits for the Voronoi region index.

[1] $atomic\_min(addr, val)$ reads word old located at the address $addr$, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction [16].

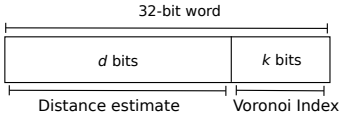


Fig. 3. Encoding information of distance and Voronoi index in a single word. Values d and k can be changed to adjust precision.

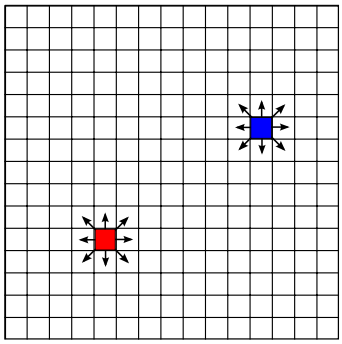


Fig. 4. Scatter updates: each active thread propagates its current information about distance and Voronoi index to its neighbors.

A. Data structure

Our data representation in memory substantially differs from the classical graph data structures. Instead of using adjacency matrices or lists for the shortest path computation, like in [2], we are dealing directly with images which are 3D node matrices. Each node keeps its compliance value, Voronoi cell index, and distance to a Voronoi source. The connectivity between nodes is given by their natural neighbors in the 3D volume (26 neighbors). The weight of each edge is given by some generalized distance function, $dist(C_v, C_u)$, relating adjacent neighboring nodes. In this work specifically, we employ the compliance scaled distance function used by Faure et Al. [1]. In this case, the distance between two adjacent nodes is a function of the measure of compliance of the material at each node.

B. Base algorithm

Our algorithm uses four internal arrays, C_0, C_1, Vor and $Mask$, stored on the GPU global memory and with same size of the input volume (Fig. 5). The cost arrays C_0 and C_1 are used to keep the shortest-path estimates of each voxel. They are initialized with 0 at the seeds and ∞ (maximum unsigned integer value) everywhere else. The Voronoi diagram, stored on array Vor, is initially empty on every voxel, except for those corresponding to the seed's coordinate which are initialized with a unique Voronoi cell index. Finally, the boolean array $Mask$ is used as activity mask to mark which voxels have an updated cost estimate indicating that it will be relaxed on the next step.

We assign one thread to every voxel . The execution then follows a scatter approach (Fig. 4) where each active thread, marked on $Mask$, will relax the cost estimate of its neighbors and set their correct Voronoi index.

The algorithm is divided in two parallel phases: relaxation

```
1: procedure VORONOI(Seeds, Vor, Mat)
2:     for all v ∈ Mat do
3:         C0[v] ← ∞; C1[v] ← ∞
4:     end for
5:     for all s ∈ Seeds do
6:         C0[s] ← 0; C1[s] ← 0
7:         Mask[s] ← true
8:         Vor[s] ← idx++
9:     end for
10:    repeat
11:        RELAXKERNEL(Mask, C0, C1, Mat)
12:        TERM ← true
13:        UPDATEKERNEL(Mask, C0, C1)
14:    until TERM
15: end procedure
```

Fig. 5. Host Code

```
1: procedure RELAXKERNEL
2:     tid ← getThreadIndex()
3:     if Mask[tid] then
4:         for all neighbors nid of tid do
5:             d_new ← C0[tid] + localDist(tid, nid, Mat)
6:             AtomicMin((C1[nid] | Vor[nid]),
7:                                   (d_new | Vor[tid]))
8:         end for
9:         Mask[tid] ← false
10:    end if
11: end procedure
```

Fig. 6. Relaxation Kernel: updates the current shortest path estimates and the closest Voronoi seed.

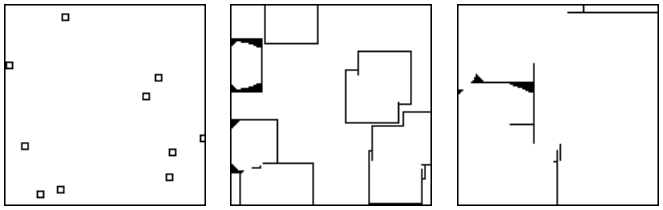
and update. The host code (Fig. 5) initializes the data structures and then iteratively calls the GPU kernels RELAXKERNEL (Fig. 6), add UPDATEKERNEL (Fig. 7), until the termination condition is satisfied. The distance function, at line 5 in RELAXKERNEL, computes the local distance between two neighbor voxels based on their compliance values in the material map [1]. At each iteration, C_1 maintains the intermediate values computed during the relaxation. In the UPDATEKERNEL procedure, the values from C_1 are copied back to C_0 and the activity mask is updated. The duplication of these cost matrices is needed to avoid read-after-write hazards when writing to global memory. The algorithm finishes when the diagram reaches a fixed point, where no more voxels are updated.

C. Algorithm with stream compression

As an enhancement to our base algorithm, we tried to reduce the number of idle threads by applying stream compression [2], [17]. Fig. 8 shows how the computation propagates to neighbors in a form similar to a wave. The black front indicates, at each step, which threads have *True* in the activity mask at the beginning of the relaxation kernel (at line 3 of algorithm in Fig. 6). Over the execution of the algorithm, we note that the number of active threads is much lower than the

```
1: procedure UPDATEKERNEL
2:     tid ← getThreadIndex()
3:     if C0[tid] > C1[tid] then
4:         C0[tid] ← C1[tid]
5:         Mask[tid] ← true
6:         TERM ← false
7:     end if
8: end procedure
```

Fig. 7. Update kernel: verifies the termination condition and updates the activity mask.



(a) step 3 (b) step 17 (c) step 42

Fig. 8. At each step the thread activity mask is updated. This process triggers propagation waves leaving from each Voronoi seed. As the distances are not linear some pixels will be recomputed causing the effect of "thicker waves" (b).

grid size and also varies considerably along time (Fig. 12). This causes our thread blocks to be very inefficient as most of the threads will actually evaluate the conditional to *False*, without computing anything (line 3, Fig. 6) .

Compact blocks solve this problem by grouping all the active thread in fewer blocks, thus reducing branch divergence, as well as the runtime overhead of scheduling idle threads.

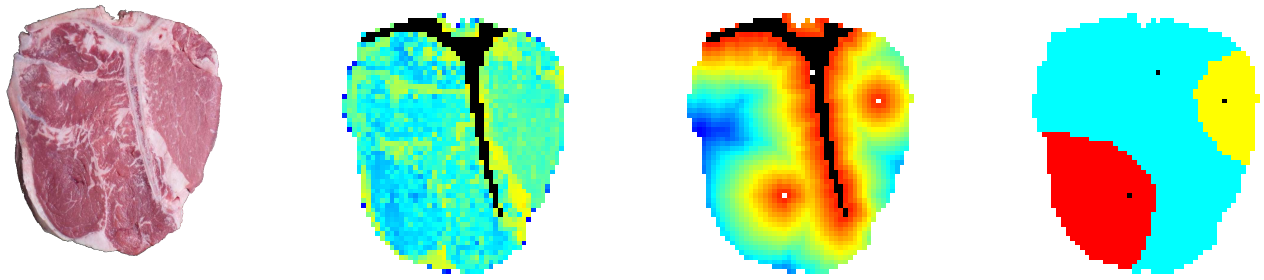
Implementation: Stream compression is performed by a *Scan* operation over the activity mask followed by a *Scatter*. These operations can be easily implemented in CUDA using the Thrust template library [18]. The result of the compression is an array mapping thread indexes to pixel coordinates, that are used in Kernel1 to retrieve the correct data.

This process adds a non-negligible overhead which sometimes can actually supersede the gains of performance. To be able to balance the trade-off between performance gains of compression and time spent by the scan+scatter process, we implemented a variable grain compression.

Variable grain compression: This method defines a coarser subdivision over the activity mask as show in Fig. 10. The coarser mask is parametrized by its grain size, which is set by their x, y and z dimensions. The algorithm then scans the coarser mask, identifying which grains contain active threads and launches only this amount of threads. We will use the notation $dim_x \times dim_y \times dim_z$ to refer to different grains used.

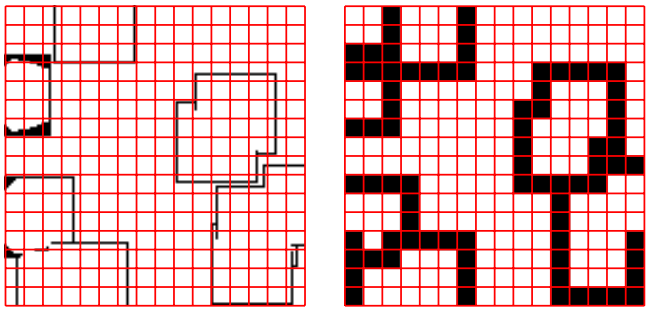
IV. EXPERIMENTS

Several benchmarks were performed to evaluate the performance of our algorithm. In the following sections we describe our test environment and input instances used for the experiments.



(a) T-bone Steak (b) Stiffness (c) Compliance-scaled distances (d) Voronoi Partitions

Fig. 9. **Use case example:** The T-bone steak (a) contains a mixture of flexible meat, softer grease and a rigid bone. As input we take the voxelized material map of stiffness values (b) and the coordinates of the simulation nodes. Our method computes the Voronoi diagram (d) rooted at each node using a compliance-scaled distance metric (c).



(a) Activity Mask subdivision (b) Coarser Mask

Fig. 10. Stream compression with variable grain size.

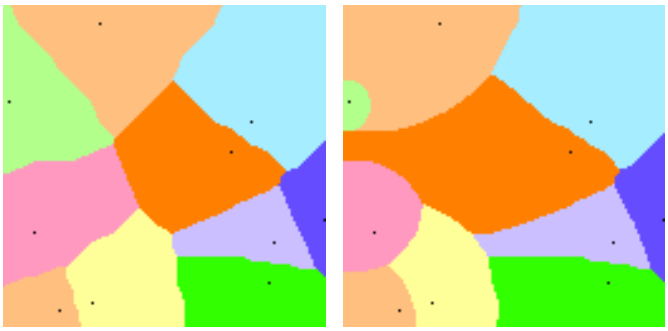


Fig. 11. Comparison of Voronoi diagrams generated with the same set of seeds on two different material maps. Left: with an uniform stiffness. Right: with a stiffness gradient.

A. Testing environment

The platform used for the CPU benchmarks was an Intel CoreTM i7 CPU model 930 with 4 cores running at 2.89Ghz and 12 GB memory. Despite the multi-core architecture, the CPU implementation is strictly sequential. The results of our GPU algorithm were obtained on an NVIDIA GPU GTX480 with 1.5 GBytes of global memory and 15 Multiprocessors with 32 cores each, totaling 480 CUDA cores. The CPU codes were compiled with GCC 4.8 using -O2 optimization flags. The CUDA driver is version 6 while the run-time is version 5.5.

B. Input instances

The input data set used differs on 3 different parameters: volume size, material map topology and number of Voronoi seeds. For the material map topologies we considered both synthetic and real-application data. The synthetic topologies represents a cube volume with (a) an uniform constant stiffness and (b) a gradient stiffness varying uniformly from left to right (called *Gradient*-Fig. 1a). In these topologies, we variate the volume from 32^3 to 256^3 voxels, which are the common discretization sizes used for physics simulation in [1]. The seeds are randomly distributed on each map. We note that, due to the compliance-scaled distance function employed, the same set of seeds actually generate very different Voronoi diagrams, depending on the topology of the material map used (see Fig. 11).

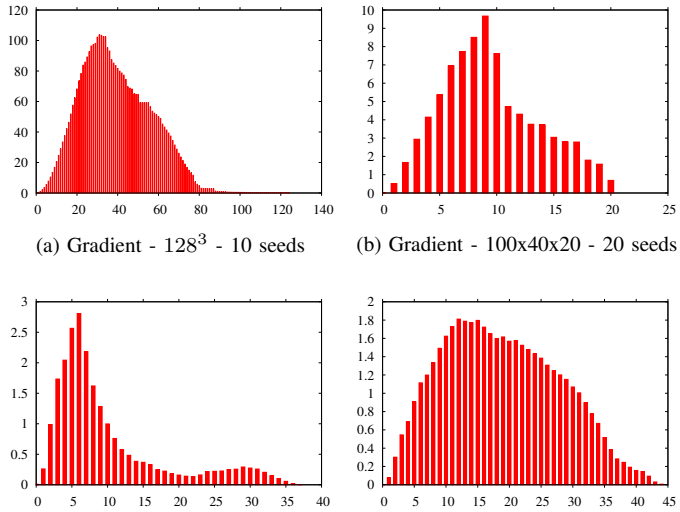
The real-application data-set is the discretized material map of the T-bone steak (Fig. 9) from paper [1]. The map of the steak has a volume of size 64x64x15 voxels and exhibits non-uniform stiffness distribution. The data of the steak is freely available for download with the SOFA framework [19], [20].

V. Results and Discussion

This section presents the results obtained on several test cases. For a proper analysis, we divide our experimental results in three parts presented bellow.

A. Base algorithm speed-up

We start by comparing our parallel algorithm with its sequential reference implementation on CPU. The results are shown in the form of parallel speed-ups on Fig. 13. Each bar represents a different input instance where labels *cube*32^3*10s*, *cube*64^3*10s*, *cube*128^3*10s* and *cube*256^3*10s* denote a cube with gradient topology with dimensions 32, 64, 128 and 256 respectively, each with 10 seeds. The *plate 100x40x10 20s* input is a plate of stiffness gradient with 20 seeds (shown in Fig. 1). Both *steak* instances have a bounding volume of 64x64x15 voxels.



(a) Gradient - 128^3 - 10 seeds (b) Gradient - 100x40x20 - 20 seeds

(c) Steak - 64x64x15 - 10 seeds (d) Steak - 64x64x15 - 3 seeds

Fig. 12. Thousands of active threads at each iteration of RELAXKERNEL.

In this benchmark the speedup achieved varies from 3.8x for small volumes (Steak) up to almost 40x for bigger ones. These results show that our algorithm benefits from bigger input sizes, because they expose more parallelism. This is also confirmed in Fig. 12, where we note that the maximum amount of active parallel threads is higher for bigger volumes. We summarize the execution times obtained with this benchmark in Table I .

B. *Voronoi seeds*

On a second scenario, we investigated the impact of the number of seeds in the performance. We used a volume size of 128^3 and same gradient topology. For each quantity of seeds, we randomly generated 10 different seed sets. Each result shown on Fig. 14 is the average speed-up obtained with these 10 instances. These results suggest that for larger amounts of seeds the speedup increases. Indeed, having more Voronoi seeds has the effect of allowing more active threads at the first iteration. Moreover, the number of iterations of the algorithm tends to reduce as more Voronoi cells expand concurrently.

C. *Stream compression*

Our last set of experiments evaluates the stream compression optimization described in section III-C. This optimization can be parametrized by setting the grain dimensions used for the subdivision of the coarser mask like shown on Fig. 10. We used the CUDA profiling tools to analyze the trade-off between overhead of stream compression and gained performance at several grains. We summarized the most representative results in the stacked histogram of Fig. 15. The figure presents results for a volume size of 128^3 with gradient topology and a set of 10 fixed seeds. The bars are sorted by total execution time and each grain size is indicated on the *x* axis, where "static" refers to the base algorithm.

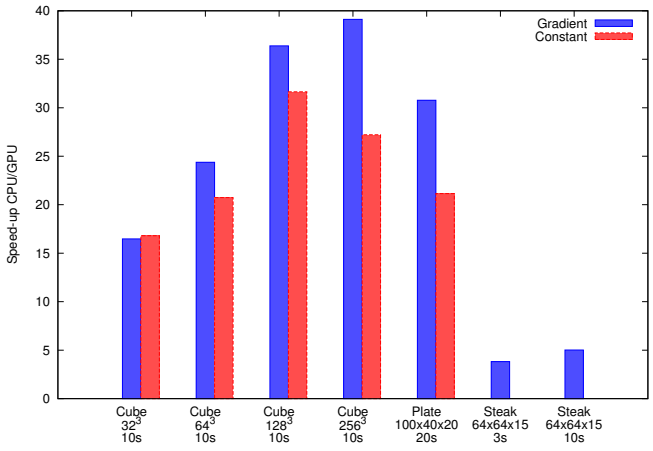


Fig. 13. Speed-up for different input sizes. Gradient and constant topologies are presented for synthetic benchmarks only. Steak's topology corresponds to the real data-set of Fig. 9

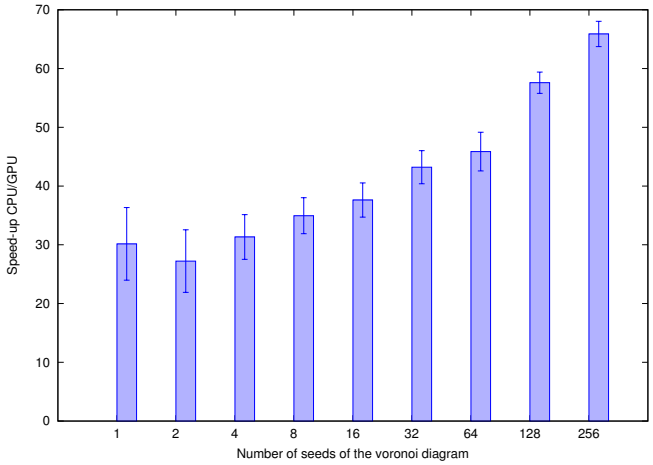


Fig. 14. Average speed-up when increasing the number of seeds of the Voronoi diagram. Standard deviations are shown on top of each bar.

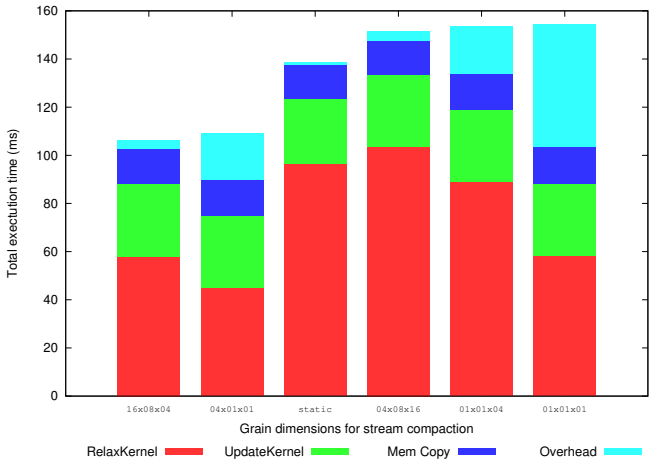


Fig. 15. Profiling of stream compression

TABLE I
BENCHMARK RESULTS WITH EXECUTION TIMES AND SPEED-UP OBTAINED WITH THE BASE GPU ALGORITHM.

			CPU		GPU		
Topology	**Volume**	**#Seeds**	**Iterations**	**Time (ms)**	**Iterations**	**Time(ms)**	**Speed-up**
Gradient	32^3	10	32768	32.24	30	1.96	16.46
	64^3	10	262144	296.77	70	12.13	24.46
	128^3	10	2097152	3863.54	126	105.91	36.48
	256^3	10	16777216	38756.80	195	985.80	39.31
	$100 \times 40 \times 20$	20	80000	77.04	21	2.51	30.68
Constant	32^3	10	32768	21.37	20	1.27	16.80
	64^3	10	262144	190.81	52	9.20	20.73
	128^3	10	2097152	1957.65	79	61.85	31.65
	256^3	10	16777216	22103.64	159	812.03	27.22
	$100 \times 40 \times 20$	20	80000	52.31	22	2.47	21.15
Steak	$64 \times 64 \times 15$	3	12276	10.80	38	2.16	5.01
	$64 \times 64 \times 15$	10	12276	10.39	45	2.72	3.82

Finner grains generate larger masks, therefore add more overhead for generating the map of active threads. A finner grain, however, provides better compression, which reduces the amount of idle threads and of useless thread-blocks. This positively impacts the time spent on the RELAXKERNEL procedure. See column $1 \times 1 \times 1$ in Fig. 15.

On the other-hand, more compact thread-blocks will also increase the number of threads accessing non-coalesced memory locations. In our application the volume is stored linearly in memory, which means that neighbour voxels on the x dimension are stored contiguously in memory. Favoring a larger x grain-dimension increases the number of threads accessing the same memory segment, thus achieving a better memory throughput. This fact can be observed comparing grains of same sizes, but different shapes like $4 \times 1 \times 1$ and $1 \times 1 \times 4$.

Experiments with smaller volume sizes, like 32^3 and 64^3, showed worse total execution time than the base algorithm. Nevertheless, for the 128^3 volume, the technique of stream compression led to a 23.29 % performance gain over the base implementation.

VI. CONCLUSION AND FUTURE WORK

In this work we presented a GPU algorithm for computing the Voronoi diagram with generalized distance functions. Our method adapts a graph algorithm, for the SSSP problem, to compute the Voronoi diagram on a 3D grid of voxels. We have shown through experimental evaluation that our base parallel implementation significantly speeds-up Voronoi computation. Additionally, we applied an optimization strategy called *Stream compression* that allows to increase utilization of the GPU on large volumes.

Regarding our implementation, there is still room for optimization on the data representation in memory. Its current linear representation cannot benefit from the locality present in the neighborhood computation. A better memory layout, like Z-curves, could further enhance the performance.

Our algorithm has a direct application on physics-based simulation algorithms. As a next step towards this direction, we plan to address the dynamic scenario simulating cuts in deformable objects. In this case, local updates of the Voronoi diagram would be needed to handle dynamic changes in topology.

ACKNOWLEDGMENT

We would like to thank Françcois Faure from INRIA-Grenoble for its insightful help and discussion about the method of material-aware distance function. We also thanks CAPES and CNPq (process 476685/2012-5 and 309483/2011-5) for the financial support.

REFERENCES

[1] F. Faure, B. Gilles, G. Bousquet, and D. K. Pai, "Sparse meshless models of complex deformable solids," *ACM SIGGRAPH 2011 papers on - SIGGRAPH '11*, p. 1, 2011.

[2] P. Harish, V. Vineet, and P. J. Narayanan, "Large Graph Algorithms for Massively Multithreaded Architectures," International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74, 2009.

[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[4] G. Rong, Y. Liu, W. Wang, and X. Yin, "GPU-assisted computation of centroidal Voronoi tessellation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 3, pp. 345–356, 2011.

[5] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Fast computation of generalized Voronoi diagrams using graphics hardware," *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 277–286, 1999.

[6] G. Rong and T. Tan, "Jump flooding in GPU with applications to Voronoi diagram and distance transform," *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, p. 109, 2006.

[7] M. Erwig, "The graph Voronoi diagram with applications," *Networks*, vol. 36, no. 3, pp. 156–163, 2000.

[8] F. Hurtado, R. Klein, E. Langetepe, and V. Sacristán, "The weighted farthest color Voronoi diagram on trees and graphs," *Computational Geometry*, vol. 27, no. 1, pp. 13–26, Jan. 2004.

[9] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel, "Parallel algorithms for approximation of distance maps on parametric surfaces," *ACM Transactions on Graphics*, vol. 27, no. 4, pp. 1–16, Oct. 2008.

[10] D. Reem, "On the possibility of simple parallel computing of Voronoi diagrams and Delaunay graphs," *CoRR*, vol. abs/1212.1, pp. 1–31, 2012.

[11] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of Dijkstra's shortest path algorithm," *Mathematical Foundations of Computer Science 1998*, pp. 722–731, 1998.

[12] S. Kumar, A. Misra, and R. S. R. Tomar, "A modified parallel approach to Single Source Shortest Path Problem for massively dense graphs using CUDA," *2011 2nd International Conference on Computer and Communication Technology (ICCCT-2011)*, pp. 635–639, Sep. 2011.

[13] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs," *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units - GPGPU-6*, pp. 96–107, 2013.

[14] K. Madduri, D. Bader, J. Berry, and J. Crobak, "Parallel shortest path algorithms for solving large-scale instances," in *9th DIMACS Implementation Challenge – The Shortest Path Problem*, DIMACS Center, Rutgers University, Piscataway, NJ, 2006, pp. 1–39.

[15] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "A new GPU-based approach to the shortest path problem," *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pp. 505–511, Jul. 2013.

[16] NVIDIA, "Cuda c programing guide," "http://docs.nvidia.com/cuda/cuda-c-programming-guide/", 2014, [accessed on 23-Apr-2014].

[17] J. Hoberock, V. Lu, Y. Jia, and J. Hart, "Stream compaction for deferred shading," *Proceedings of the Conference on High Performance Graphics*, vol. 1, no. 212, pp. 173–180, 2009.

[18] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU Computing Gems*, vol. 7, 2011.

[19] P. SOFA, "Simulation open framework architecture," "http://www.sofa-framework.org/", 2014, [accessed on 23-Apr-2014].

[20] F. Faure, C. Duriez, H. Delingette, J. Allard, B. Gilles, S. Marchesseau, H. Talbot, H. Courtecuisse, G. Bousquet, I. Peterlik, and S. Cotin, "SOFA: A Multi-Model Framework for Interactive Physical Simulation," in *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*, Y. Payan, Ed. Springer, Jun. 2012.