Parallel Voronoi Diagram computation on scaled distance planes using CUDA

Julio Toss, João Comba
Institute of Informatics - Federal University of Rio Grande do Sul (UFRGS)
Av. Bento Gonalves 9500, Porto Alegre - RS - Brasil, CEP: 91501-970
{jtoss,comba}@inf.ufrgs.br

Abstract

Voronoi diagrams are fundamental data structures in computational geometry with several applications on different fields inside and outside computer science. This paper shows a CUDA algorithm to compute Voronoi diagrams on a 2D image where the distance between points cannot be directly computed in the euclidean plane. The proposed method extends an existing Dijkstra-based GPU algorithm to treat our 2D images as graph and then compute the shortest-paths to create each Voronoi cell. Experimental results report speed-ups up to almost 40x over current reference sequential method for Voronoi computation on non-euclidean space. This problem is a building block of the deformation engine in the SOFA physics simulation framework.

1. Introduction

Voronoi Diagrams appears in many fields of computer science. Classical problems like *Finding Nearest Site, Facility Location, Motion Planning or Coverage in sensor networks*, in general use some kind of Voronoi diagrams underlying its solutions.

This paper will address Voronoi Diagrams in the context of the physics based simulation method proposed in [1] to compute the deformations of complex objects. Actually, in this method, the simulated bodies are composed with a mixture of soft and stiff materials so, to have an accurate simulation, each simulation node has to limit the region of influence where they will propagate the forces. These regions are, in fact, represented as Voronoi cells.

A particularity of this kind of Voronoi diagram is that the distance function is not computed in a standard Euclidean plane. Instead, the distance between points is scaled according to the compliance values in the material map. Therefore, points connected by similar material (i.e. inside a same Voronoi cell) will deform in a similar way.

The creation of the Voronoi Diagram is done during the setup phase of the simulation and remains unchanged during the simulation as long as the object's topology and material property values do not change. This allows the following simulation phase to be performed in real-time, a necessary requirement for interactive applications.

However, to enable on-line modifications on the object's topology, we must be capable of recomputing the Voronoi Diagrams on-the-fly, that is, during the simulation. Currently, the time needed to recompute Voronoi Diagrams is too costly for a real-time simulation. In [1] the authors report initialization times ranging from less that 1 second for grid of 100x40 voxels to 10 minutes for a 500x200 grid. However, their implementation in strictly sequential leaving plenty of room for parallelization.

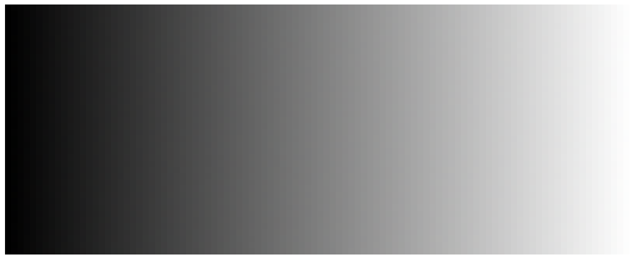
This paper proposes a parallel method for computing the afore mentioned kind of Voronoi Diagram. Our method, described in section 2, is based on the Dijkstra-like CUDA algorithm proposed in [2, 3] for solving the Single-Source-Shortest-Path (SSSP) problem on large graphs .

2. Parallel Voronoi Implementation

As mentioned previously, the kind of Voronoi diagram our method computes differs from other works [4, 7, 6] because we are not in a classical euclidean space. The distances within our space are scaled according to the stiffness values of the material. The distance function is locally computed between neighbor pixels using the euclidean distance and then scaled by the corresponding values from W. The distance function is defined by equation 1, where W is a matrix containing the material stiffness values.

$$d(p,q) = \frac{\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} * 2}{W_p + W_q} \tag{1}$$

To determine the distance between pixels in different neighborhoods we compute the shortest path on a graph where each pixel is mapped to a vertex and connected to its neighbors by weighted edges with weights given by $d(p,q)$.


(a) Input material map with a stiffness gradient

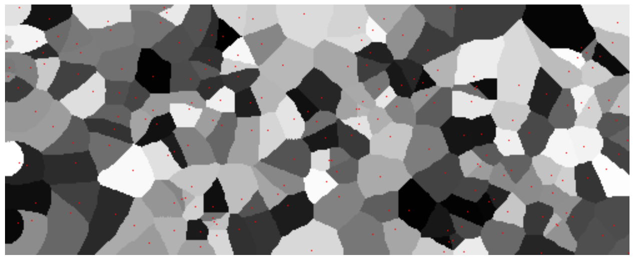

(b) Voronoi diagram for a 500x200 pixels grid and 200 seeds (red pixels)

Figure 1: Example of input material property map (1a) and the corresponding Voronoi diagram generated (1b)

2.1. Pseudocode

Our algorithm is inspired on the SSSP problem solution described in [3, 5]. The algorithm executes Kernel1 followed by kernel2 until the termination condition is satisfied. At each iteration, two cost matrices, C_o and C_u, are updated to keep the cost of the shortest path found so far. The duplication of these cost matrices is needed to avoid read after write inconsistencies when writing to global memory. The call to the distance function at line 4 of Kernel1 accesses the W matrix containing the stiffness values of each point. At last, the VOR matrix stores the index of the nearest Voronoi seed of each pixel. A boolean matrix M is used to mark which vertex have changed and will need to be relaxed on the next step. Initially only the seed vertexes are marked. When the graph has reached an equilibrium where no more vertex is updated, the algorithm finishes.

3. Performance Evaluation

Benchmarks were performed with two kinds of material maps. One constant, where all the pixels had the same value of stiffness and the other with a gradient variation of stiffness like shown on figure 1a. The bench instances variate image size and number of seeds. The images with sizes 32x32, 32x64 and 64x64 have all 5 seeds. The ones with pixel resolution 100x40 and 500x200 have 10 and 200 seeds respectively.

Algorithm 1 Kernel1 - Relaxation

```
1: $tid \leftarrow getThreadIndex()$
2: if $M[tid]$ then
3:     for all neighbors $nid$ of $tid$ do
4:         $newCost \leftarrow C_o[tid] + d(tid, nid)$
5:         AtomicMin($C_u[nid]$, $newCost$)
6:         if $newCost = C_u[nid]$ then
7:             $VOR[nid] \leftarrow VOR[tid]$
8:         end if
9:     end for
10:    $M[tid] \leftarrow False$
11: end if
```

Algorithm 2 Kernel2 - Verify termination and update

```
1: $tid \leftarrow getThreadIndex()$
2: if $C_o[tid] > C_u[tid]$ then
3:     $C_o[tid] \leftarrow C_u[tid]$
4:     $M[tid] \leftarrow True$
5:     $CONTINUE \leftarrow True$
6: end if
7: $C_u[tid] \leftarrow C_o[tid]$
```

3.1. Test Environment

The platform used for the CPU benchmarks was an Intel Core^{TM}i7 CPU model 930 with 4 cores running at 2.89Ghz and 12 GB memory. However, despite the multi-core architecture, note that the CPU implementation is strictly sequential. The results for 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 installed CUDA Driver and Runtime were of version 5.0.

3.2. Results

The results reported in this section are an average of 10 runs measuring time (ms) and are summarized in table 1. A comparison of performance based on speed-up obtained with the GPU version over the sequential CPU implementation is shown on figure 2.

The results show that the GPU implementation is always better than the sequential CPU version, even for small images, but it becomes much more interesting for large images. As expected, the speed-up increases with the size of the input image. Because each pixel is computed by one CUDA thread more parallelism is exposed in high resolution images.

Despite the good speed-up achieved, if we look closer at table 1 we realize that the method has a very low efficiency. From 100,000 threads used for the 500x200 benchmark, the average of active threads actually doing work per iteration

			CPU			GPU		
Topology	**Size**	**#Seeds**	**Iterations**	**Time (ms)**	**Std. Dev**	**Iterations**	**Time (ms)**	**Std. Dev**
Degrade	32 x 32	5	1024	5.35	1.393	18	0.642	0.002
	64 x 32	5	2048	8.323	2.668	32	1.164	0.004
	64 x 64	5	4096	17.561	3.283	62	2.373	0.01
Uniform	32 x 32	5	1024	4.743	1.144	20	0.707	0.002
	64 x 32	5	2048	7.671	1.921	25	0.918	0.006
	64 x 64	5	4096	15.369	3.485	46	1.743	0.008
Degrade	100 x 40	10	4000	14.45	0.059	48	1.842	0.007
	500 x 200	200	100000	381.847	5.222	54	9.585	0.019

Table 1: Benchmark results

is of only 100,000 / 54 = 1851, which means less than 2% of thread activity.

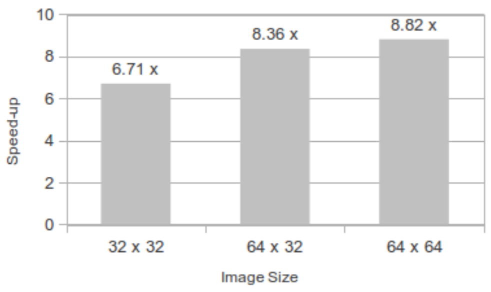
Note that the number of active thread grows at each iteration by a factor of the graph degree as new vertexes are expanded. In our case, each vertex has 8 neighbors. In practice, this number is even smaller because, as our graph has a 2D grid topology, two neighbor vertexes share half of their neighborhood thus reducing the number of new vertexes expanded at each step. Voronoi diagrams with a larger quantity of seeds will expose parallelism faster than the others because more vertex will be expanded at the initial steps.

Additionally the solution showed on algorithm 1 and 2 have an implicit global barrier at the end of each kernel call. Overheads of several successive kernel launches are known to be harmful to the overall performance. Future implementations will try to eliminate the need of the global barrier, a good candidate solution could be based on the modified Bellman-Ford algorithm proposed in [5].

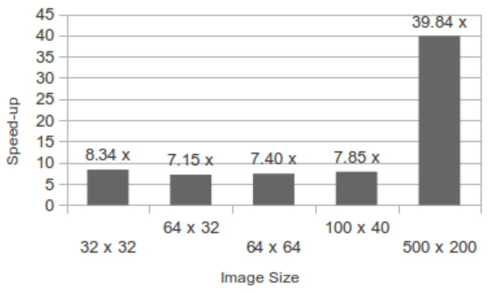
4. Conclusions & Future Works

Good parallel efficiency on general purpose graphics processor is very hard to achieve. The SIMD nature of such platform require that threads execute rather synchronized with little divergence on its execution paths. This is particularly harmful for graph algorithms, like Dijkstra SSSP, where the expansion of vertexes may lead to very unbalanced workloads. For future improvement, load-balancing techniques could be used to assign vertex to threads in a way that reduces the number of idle threads scheduled at each iteration.

Targeting synchronization reduction, our next implementation will be based on the algorithm proposed by [5] were we remove the global barrier at the expense of introducing some shortest paths re-computation.



(a) Uniform stiffness image



(b) Gradient stiffness image

Figure 2: Speedups of the GPU implementation compared to CPU.

Finally, regarding the simulation framework, there are many other problems beyond Voronoi diagrams that could benefit from parallelization. Also, future implementation could target heterogeneous architecture combining multicores CPUs and Multi-GPUs.

4.1. Acknowledgement

Special thanks to François Faure and Bruno Raffin from INRIA-Grenoble for their insightful help and explanation about the method of material-aware distance function computation.

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*, page 1, 2011.

[2] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *High Performance ComputingHiPC 2007*, pages 197–208, 2007.

[3] P. Harish, V. Vineet, and P. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, 2009.

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

[5] 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)*, pages 635–639, Sept. 2011.

[6] G. Rong, Y. Liu, W. Wang, and X. Yin. GPU-assisted computation of centroidal Voronoi tessellation. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):345–356, 2011.

[7] 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 ...*, page 109, 2006.