Ezra Davis
ezra.davis@yale.edu
Parallel Programming CPSC 524
Andrew Sherman
Final Course Project
Spring 2018
May 15, 2018

Re-implementation of *Procedural Voronoi Foams for Additive Manufacturing*

Preface

In the attached appendix, I am including a companion paper for background information on analysis and fabrication of 3D printed Voronoi foams of varying densities entitled, "Analysis of *Procedural Voronoi Foams for Additive Manufacturing,"* which I wrote for a final course project for Computational Complexity CPSC 568. I have included this to provide a concise introduction to the topic and avoid repetition.

1. Introduction

I implemented the three core algorithms described in the paper "Procedural Voronoi Foams for Additive Manufacturing" [Martínez et al. 2016], which is an embarrassingly parallel algorithm for determining a 3D open cell Voronoi diagram in preparation for 3D printing. It performs a calculation for each voxel in the final 3D printable product. Both I and the Martínez et al. [2016] implemented the algorithms using CUDA C, an API for a GPU compute device..

For a previous research project I implemented these three algorithms serially in Python. My Python implementation using the serial approach resulted in very slow performance. I then pursued a more traditional serial Voronoi diagram algorithm (using the qhull library); this second iteration proved to be significantly faster, but still quite slow performance overall. My difficulties with that project is what motivated me to pursue further study in both parallel computing and computational complexity. I hoped to be able to take my newly learned understanding of approaches and techniques and then revisit this same problem of 3D printable Voronoi foam of varying densities.

In this endeavor I have re-implemented these algorithms using a different language (CUDA C++) in an attempt to increase speed and efficiency.

2. Coding approach, changes, and challenges

Approach

To approach the coding project, I examined Martínez et al.[2016] pseudocode in detail, and then wrote my own version of the code in C++, which can also compile to CUDA C++.

The 2016 paper breaks the process of creating a Voronoi foam into three algorithms: generating Voronoi seeds in a small volume (`subdivide_cell`), determining which of those volumes could influence any particular point (`gather_seeds`), and finally using those Voronoi seeds to determine if a query point/voxel should be part of a foam. I made a few changes to the original algorithms, especially `gather_seeds`.

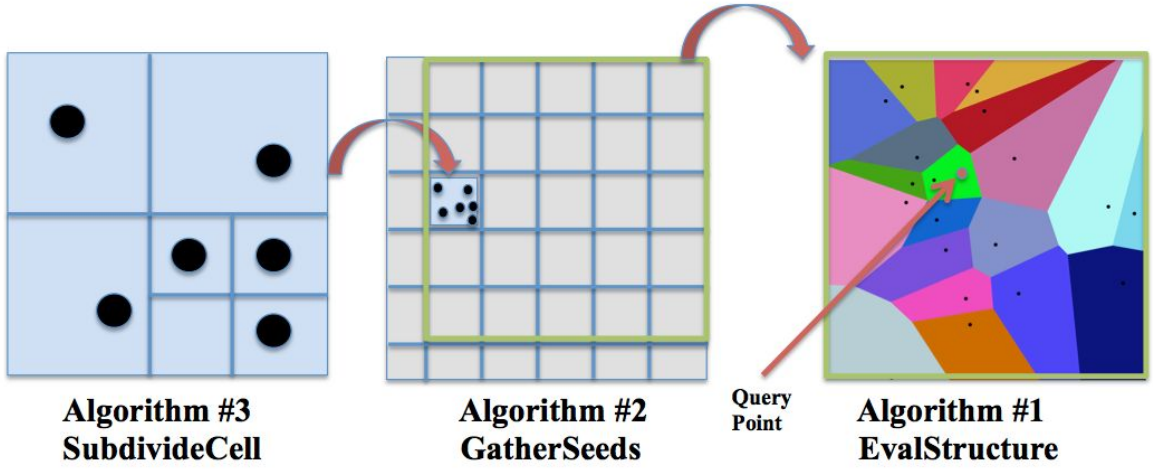

Figure 1. Illustration of the three algorithms used to create the Voronoi foam.
Algorithm #3 is first used to determine the Voronoi seeds in a grid cell.
Algorithm #2 is used to determine which grid cells could possibly affect our query point.
Algorithm #1 determines if the query point is in a beam of the Voronoi foam.

Euclidean Voronoi diagram illustration courtesy of Ertl [2015]

Streamlining `gather_seeds`

In the original paper `gather_seeds` gets a large superset of the Voronoi seeds that could influence the query point. It takes advantage of the fact that the problem assumes that there must be at least one Voronoi seed returned by `subdivide_cell`, which means that a particular point cannot be influenced by a Voronoi seed arbitrarily far away. `gather_seeds` first

determines the closest Voronoi seed to the query point, which must be in the query point's grid cell's two-ring neighborhood. The closest Voronoi seed determines the Voronoi cell that contains the query point. `gather_seeds` then finds all the Voronoi seeds that determine the borders of that Voronoi cell. This seems logical, but because determining the borders of the query point's Voronoi cell allows us to determine if the query point is on the border of the cell later on. The points that could influence a Voronoi cell must be in a two-ring neighborhood of its center.

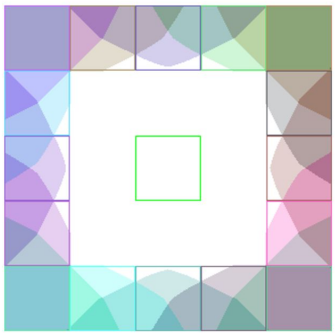

Figure 2. A 2D cross section of a two-ring neighborhood of a grid cell. A Voronoi seed in the cell outlined in green could be influenced by a point in any of the non-shaded areas. [Martínez et al., 2016]

Each two ring neighborhood contains 5^3 (or 125) grid cells, and thus necessitates 125 calls to `subdivide_cell`. In total, `gather_seeds` can perform up to 223 calls to `subdivide_cell` (one neighborhood plus another minus the overlap, which can be as little as 3^3).

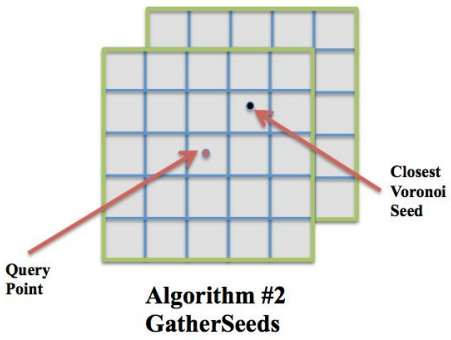

Figure 3. A possible set of grid cells collected by the algorithm

After some thinking, I realized that the Voronoi seeds added to fully specify the Voronoi cell (not in the original two-ring neighborhood of the query point) didn't need to be checked - they specify the borders of the Voronoi cell far away from the query point.

Saving space in `gather_seeds`

I then implemented the algorithm with my modifications, and stored the resulting array of over 125 seeds in global memory on the GPU. Unfortunately, with even a moderate sized foam (100x100 voxels) I ran out of memory.

To solve this issue I created three options:

1. Have one thread in the block calculate the seeds, which are then shared amongst its block in either shared or main memory
2. Recalculate the Voronoi seeds on the fly
3. Limit the total number of threads and have each thread work on more voxels

Option 3 is not great because it simply does not scale well, and will just have the same problem later. Option 1 is much better but is a little finicky - I would have to make sure that the thread blocks line up with the voxels' grid cells. Due to these reasons, I went with option 2: recalculating the Voronoi seeds each time I need them. This method has a certain elegance and simplicity that reminds me of functional programming, and while potentially much slower than option 1, I wasn't overly concerned.

I created a new class `SeedGatherer` that acts as an iterator for lazily evaluating `gather_seeds`. This was the first part of my code that truly used features from C++ that aren't in C. I initially had some issues with stack overflow because I allocate up to four `SeedGatherer`s at a time, but I reduced the (unnecessarily large) buffer space allocated for each of them, and the problem solved itself.

Unfortunately, using the `SeedGatherer` class slowed down the program dramatically - taking far longer than the few milliseconds per z-layer described in the Martínez et al. paper. It was still far faster than the serial Python implementation. The slowdown is caused by the fact that each seed returned by `SeedGatherer` is accessed n^3 times. Looking back on my choice, I should have gone with option 1, which would have avoided this expensive recomputation. That said, the program is still fast enough to be practical.

3. Source Code

- `code/`
  - `depthmap.py` - A Python script used to create a depthmap when given the results of the C++/CUDA program. With minor modification and the right libraries installed it can also use Marching Cubes to create an `stl` file of the foam.
  - `Makefile` - The makefile for compiling the project. Instructions on how to use it are in the "Building and running" section.
  - `proj.cu` - The main CUDA program. Currently setup to calculate a 100x100x100 voxel foam. Outputs one 2D slice of the foam at a time to `stdout`.
  - `test_voronoi_query.cpp` - A C++ file that I used to test `voronoi_query.cpp` without access to a working CUDA installation.
  - `voronoi_query.cpp` - The core code for the algorithm. Can run as both CUDA and C++ code.
  - `voronoi_query.h` - The header file for `voronoi_query.cpp`, which also contains a few configuration preprocessor commands.
  - `voronoi_query_old.cpp` - The core algorithms using the version of `gather_seeds` that saves the list of Voronoi seeds into global memory. Only kept around for comparison's sake as it overfills global memory with even moderately sized problems.
- `davis_complexity_paper.pdf` - My final project for Computational Complexity, which is also based on Martínez et al. [2016].
- `depthmap.png` - An example depthmap output by `depthmap.py`.
- `final_project_proposal.txt` - The original project proposal.
- `foam.stl` - An example 3D model output by a slight modification to `depthmap.py`.
- `Martinez_et_al_paper.pdf` - A copy of the paper that this project is based on.
- `result.txt` - The output of `proj` that was used for creating `depthmap.png` and `foam.stl`

4. Building and running instructions:

The main program is intended to be run on the Grace cluster, though it isn't too picky about the particular version of CUDA or the GPU, beyond it needing to have compute capability 2.0 to handle classes.

```
make clean proj           # Compile the project
./proj > result.txt
python depthmap.py # Take result.txt and convert it to depthmap.png
```

For testing `test_voronoi_query`:

```
gcc test_voronoi_query.cpp -o test_voronoi_query
./test_voronoi_query
```

5. Further avenues for exploration

I implemented only the three core algorithms discussed in the Martínez et al. [2016] paper; there are a number of ancillary algorithms and implementation details that Martínez et al. [2016] gloss over. For instance Martínez et al. [2016] barely mention the 3D printer's custom slicer, and do not examine the algorithms for avoiding the need for support material and creating an external frame for the the foam in as much detail. These are worth topics of future exploration. More simply, I could also improve the speed of `gather_seeds` by sharing the Voronoi seed calculation across a CUDA thread block.

6. Appendix

Davis, E. (2018). Analysis of Procedural Voronoi Foams for Additive Manufacturing, Computational Complexity CPSC 568, Final Course Project , Spring 2018

Martínez, J., Dumas, J., Lefebvre, S. (2016). Procedural Voronoi Foams for Additive Manufacturing. *ACM Trans. Graph.* 35, 4, Article 44 (July 2016), 12 pages. DOI = 10.1145/2897824.2925922 http://doi.acm.org/10.1145/2897824.2925922

7. References

Ertl, B. (2015). Euclidean Voronoi diagram, via Wikimedia Commons [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)], Retrieived 2018 from https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg

Martínez, J., Dumas, J., Lefebvre, S. (2016). Procedural Voronoi Foams for Additive Manufacturing. *ACM Trans. Graph.* 35, 4, Article 44 (July 2016), 12 pages. DOI = 10.1145/2897824.2925922 http://doi.acm.org/10.1145/2897824.2925922