Parallel Delaunay Triangulation

Harvey Zhang
Carnegie Mellon University
`chengraz@andrew.cmu.edu`

Connie Fan
Carnegie Mellon University
`cfan1@andrew.cmu.edu`

1 Summary

In this project, we explored the parallelization of Delaunay Triangulation algorithms using CUDA on the GPU. Specifically, we analyzed the performance of four GPU implementations based on Voronoi diagrams. For sufficiently large input sizes, our Jump Flood GPU implementation achieves a significantly higher speedup over a single-threaded CPU Randomized Incremental implementation. For an input set of 10 million points, our best implementation has a 70$\times$ speedup.

2 Background

The Delaunay Triangulation (DT) is a specific triangulation over a set of vertex points such that every triangle satisfies the Delaunay constraint. In particular, given any triangle in the resulting triangulation, no other vertices are within the circumcircle formed by the triangle. Because of this constraint, the Delaunay Triangulation of a set of vertices avoids long, skinny triangles by maximizing the minimum angle of all triangles. It is interesting to note that there must exist one unique Delaunay Triangulation over a set of vertices containing no more than 3 points on the same circle. An example of Delaunay Triangulation is shown in Figure 1.

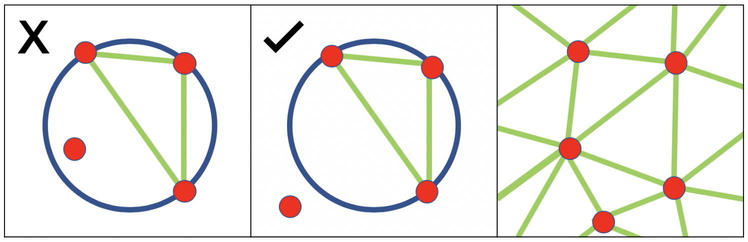


Figure 1: A triangle and point that do not satisfy the Delaunay constraint, a triangle and point that satisfies the Delaunay constraint, and an example of a Delaunay Triangulation mesh.

Generation of Delaunay Triangulation is an important task in many computer graphics applications and can also be applied to a variety of fields such as terrain modelling, path planning, and finite element analysis [1]. Because algorithms that perform Delaunay Triangulation are computationally expensive, parallelization of Delaunay Triangulation algorithms is an area of continued research.

The goal of this project is to explore parallelization of Delaunay Triangulation using the CUDA platform on Nvidia GPU. We hypothesize that the powerful support for parallel computations makes GPU a good choice for speeding up Delaunay Triangulation.

2.1 Inputs, Outputs, and Data Structures for Delaunay Triangulation

The input to the Delaunay Triangulation problem is an array of vertices $V = \{v_1, ..., v_n\}$. The output is a list of triangles $T = \{(i_1^1, i_2^1, i_3^1), (i_1^2, i_2^2, i_3^2)...\}$, represented by an array of triplets, where each element of the triplet is an index to the corresponding vertex in the list of vertices V.

For each triangle $t \in T$, we maintain the set of points that violate the Delaunay constraints $E(t)$. For example, if some point p is in triangle t's circumcircle, then $p \in E(t)$. Furthermore, some implementations also include a list of neighbors for each triangle, represented as a list of triplets where each element of the triplet is an index to the list of triangles T.

2.2 Algorithmic Approaches to Delaunay Triangulation

There are multiple algorithmic approaches to solving the Delaunay Triangulation. They are generally divided into three categories: the random incremental approach, the divide and conquer approach, and the sweepline approach [2]. We explore an approach that does not fit into any of these categories: using the Voronoi dual. Solving for the discrete Voronoi diagram over a set of vertices is the mathematical dual of solving the Delaunay Triangulation. By obtaining the Voronoi diagram, we can also solve for the Delaunay Triangulation over a set of vertices. In this project, we considered the Random Incremental Approach and the Discrete Voronoi Dual approach as potential candidates of parallelization.

2.2.1 Randomized Incremental Delaunay Triangulation

The Randomized Incremental Delaunay Triangulation algorithm works by incrementally adding points in the set of input vertices V into the existing triangulation T. At each step, we first create a valid triangulation with the newly added point v^* and then perform a series of "edge flips" to inductively maintain the Delaunay constraint [3]. The steps of the algorithm are as follows:

1. Construct a large bounding triangle around the set of input vertices V.
2. Randomize the set of vertices V to insert.
3. Randomly insert a point into the triangulation. Locate the triangle containing the point and connect the vertices of the triangle to the point.
4. Test that the Delaunay Constraint is satisfied. Perform edge flipping until the triangulation is a Delaunay Triangulation.
5. Repeat step 3 and 4 until all vertices are added to the triangulation.

The most computationally expensive phase in the Incremental Delaunay Triangulation algorithm is the Edge Flipping phase, since an edge flip between two triangles can trigger subsequent edge flips in the neighbors of both triangles. To prevent conflicts, this phase can be parallelized using fine-grained locks or fine-grained atomic instructions on vertices.

The Randomized Incremental DT also contains strong sequential dependencies. In particular, we observe that the edge flips on the same set triangles must be performed sequentially. Furthermore, after each edge flip, the Delaunay constraint of the nearby triangles must be re-evaluated since some triangles have changed orientation. This edge flip dependency leads to weak data parallelism, which may severely limit the overall parallelism of the algorithm.

2.2.2 Voronoi Dual to Delaunay Triangulation

Mathematically, the 2D Voronoi diagram is a dual of 2D Delaunay Triangulation. A Voronoi diagram partitions a plane into regions based on shortest distance to a set of points on the plane. By constructing a Voronoi diagram of the input vertices V and connecting neighboring vertices, we obtain the Delaunay Triangulation over the vertices V. Figure 2 illustrates the relationship between Voronoi diagrams and Delaunay Triangulation. Note that in the discrete case, the Delaunay Triangulation obtained from Voronoi diagram may not be exact and can be near-Delaunay.

The most computationally-expensive step in this approach is the generation of the Voronoi diagram, which is often computed over a grid of pixels and can be highly data-parallel. In particular, computations can be done nearly independently at a pixel granularity, making this approach a better candidate for the GPU than Randomized Incremental DT.

2.3 Implementation Assumptions

Throughout this project, we observe that Delaunay Triangulation over floating point vertices significantly increases the implementation complexity of many Delaunay Triangulation algorithms.

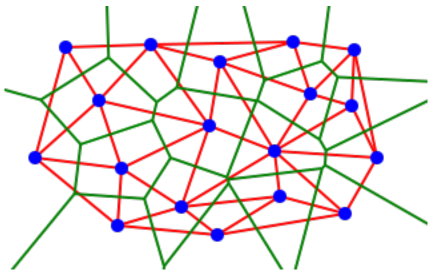


Figure 2: The 2D Voronoi diagram is a dual of 2D Delaunay Triangulation. Image from [4].

Therefore, we have decided to work with simplified Delaunay Triangulation algorithms. In particular, we assume that all input points have integer coordinates. This means that all coordinates fit onto a pixel grid and eliminates the need to handle missing and shifted points in Voronoi-based approaches.

Furthermore, we allow some results produced by the GPU implementations to be near-Delaunay (with most of the triangles satisfying the Delaunay constraint). Implementations usually have a "Delaunay Refinement" phase at the end of the algorithm, where we iterate through the vertices one last time to ensure they meet the Delaunay constraint, performing edge flipping if necessary. Algorithms for Delaunay Refinement are not within the scope of this project.

2.4 Experimental Baseline

We use the single-threaded CPU implementation based on the Randomized Incremental approach of DT provided in the Problem Based Benchmark Suite (PBBS) package [5] as the baseline to analyze our GPU implementations. It is a robust implementation of Delaunay Triangulation and a common benchmark for many DT algorithms.

3 Approach

Our parallel implementation of Delaunay Triangulation involved many iterations, each with different approaches. We began with analyzing a parallelization of the Randomized Incremental DT approach proposed by Boissonnat and Teillaud [6] in order to gain better insight into the parallelization strategies. Next, we implemented three different Voronoi-based DT approaches. Lastly, we analyzed a state-of-the-art Voronoi-based GPU implementation [7] and compared it with our GPU implementations.

Our GPU implementations are coded in CUDA and C++ and tested on an Nvidia Tesla K80 GPU.

3.1 Input Generation

We wrote a script that could generate sets of points for our input into the Delaunay Triangulation code. Parameters were number of points, the input dimension (width of possible point range), and the type of distribution (uniform or Gaussian). For uniform distributions, points were randomly generated integers in the range of 0 to $inputDimension$ for the x and y coordinates. For Gaussian distributions, points were generated about a mean of $inputDimension/2$ with standard deviation $inputDimension/4$.

Points were added to a hash set so that there were no duplicates, and were generated until the set contained the desired number of points.

Since the point coordinates are all integers, there is a relationship between the input dimension and the maximum number of points that an input set can contain. If there are a lot of points in a relatively small input grid, there is a greater likelihood that three neighboring points may be collinear, which can lead to incorrect DT (assuming no Delaunay Refinement phase). Therefore, a certain ratio between the input dimension and number of points is maintained in all test inputs.

3.2 Parallel Randomized Incremental Approach Analysis

Since we used the single-threaded Delaunay Triangulation implementation in the PBBS package [5] as our baseline, we first began our investigation by examining the parallelization strategy employed in the existing parallel CPU implementation of Randomized Incremental DT in the PBBS package. The existing parallel CPU implementation is based on OpenMP and parallelizes the edge flipping process by using fine-grained `cmpxchg` instructions.

Specifically, the algorithm maintains an array of flags F with the same size as the total number of input vertices. Points are inserted to the triangulation in parallel batches. To avoid race conditions in each batch, each vertex finds the triangle t containing it and uses `cmpxchg` to write its vertex id onto F at the indices corresponding to the vertices of t. After all points in the batch are inserted, we check the flag array F for contention between the inserted points. If contention exists, the point with higher vertex ID acquires the resources and the insert commits. The point(s) with lower vertex ID must perform a "retry" by deleting itself from the current triangulation, to be inserted again with the next batch.

We further studied this parallelization strategy by profiling the contention rate and the execution behaviors of the implementation across different numbers of threads on 10 million vertices using Intel Xeon CPUs.

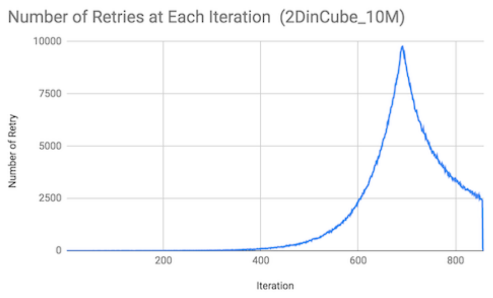


Figure 3: Number of retries at each iteration for 10 million uniformly distributed points

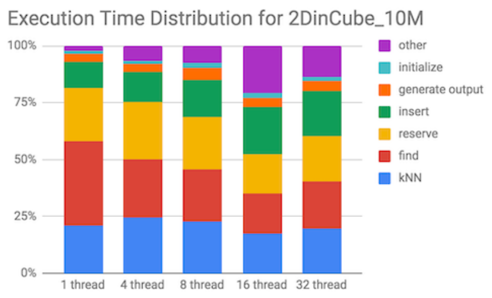


Figure 4: Number of threads vs. execution time for different phases of the randomized incremental algorithm

From the above plots, we observe that there exists heavy contention of the existing triangles between newly inserted vertices. In particular, the number of retries increases exponentially as number of iterations increases.

Although the profiling is only done on an existing CPU implementation, the timing measurements suggest that this strategy may not scale well with a large number of parallel threads. Specifically, the exponential increase in data contention produces heavy dependencies between threads and may seriously limit data-parallelism during later stages of the program execution. Thus, after analyzing this strategy, we determined that this approach may not be a suitable for the GPU and that our GPU implementation should not be based on the Randomized Incremental approach.

3.3 Voronoi Diagram-Based Approaches

After a literature review, we determined that in order to expose more parallelism in the algorithm, we must fundamentally alter our algorithmic approach. Prior work by Rong et al. [7] and Qi et al. [8] demonstrated success in generating Delaunay Triangulation from Discrete Voronoi Diagrams. Because the generation of Voronoi Diagrams can be highly data-parallel, GPU implementations can take advantage of the data-parallelism in Voronoi-based DT algorithms.

Our Voronoi Diagram-based GPU implementation consists of the following phases:

Phase 1 - Vertex Mapping: We construct a grid of pixels of size $N \times N$, where $N = 2^{ceil(\log_2(\max(x_{\max}-x_{\min}, y_{\max}-y_{\min})))}$ (max of width and height rounded to the next power of 2). We then map the input points into the pixel grid by scaling and shifting each point.

Phase 2 - Voronoi Diagram Construction: Build a Discrete Voronoi diagram from the vertices mapped on the grid.

Phase 3 - Triangle Reconstruction: Scan each pixel of the Voronoi diagram. For each pixel (x, y), look at its neighboring 2×2 region of: $(x+1, y)$, $(x, y+1)$, and $(x+1, y+1)$. If the 2×2 region contains 3 colors, generate 1 triangle. If the region contains 4 colors, generate 2 triangles as illustrated by Figure 5. We maintain a global triangle counter among all kernel threads and increment this counter using an `atomicAdd` operation in order to pack the reconstructed triangles compactly into the resulting array.

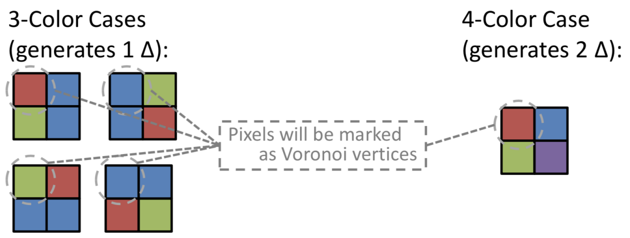


Figure 5: Each 2x2 region of pixels with 3-4 colors is marked as a vertex. Image from [9].

We observe that Phase 1 and Phase 3 of the algorithms are almost trivially parallelizable by assigning each GPU thread to a pixel. However, as shown in Figure 6, we also note that Phase 2 of the algorithm is the most computationally expensive, and efficient parallelization of Discrete Voronoi Diagrams are non-trivial. Amdahl's law suggests that parallelization efforts should be focused on the Voronoi Diagram generation phase in order to achieve the greatest speedup. Therefore, much of our implementation efforts are spent on efficient parallelization of Phase 2 of our algorithm.

3.3.1 Brute-Force Discrete Voronoi Generation

We first implemented Phase 2 of the algorithm using a brute force approach. We map each pixel to a kernel thread. Next, we compute the distance of the pixel from all the other vertices in the grid sequentially within the thread. Then, we assign the pixel to the index of the vertex closest to the pixel.

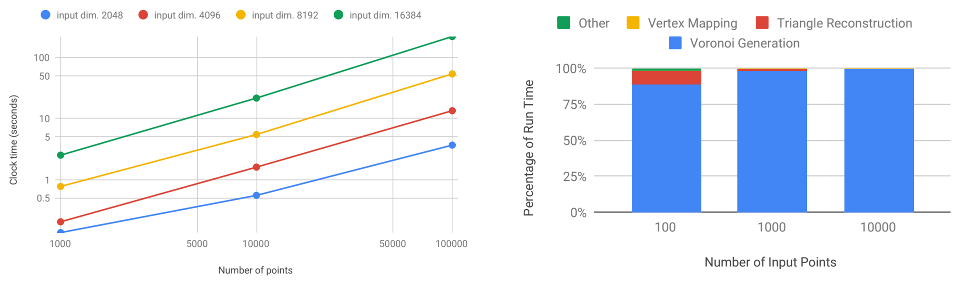


Figure 6: Total runtime and runtime distributions for the Brute Force approach with varying number of points. Both axes of the runtime plot are log-scaled. Inputs points have dimension 1024 and are uniformly generated.

We observe that this approach achieves good load balancing as each kernel thread is assigned the same number of vertices for distance computations. However, this approach had poor performance. As shown in Figure 6, runtime scales linearly with number of points across all input sizes. From the profiling results above, we notice that the Voronoi Generation phase of this approach dominates the percentage of total runtime, and the percentage of time spent on this bottleneck phase increases as the number of points increases. We hypothesize that the performance of the brute force approach is hindered due to the following reasons:

- Algorithm Complexity: Given P input points, each thread performs a minimum of $O(P)$ work for each pixel. If the number of pixels in the grid is larger than the number of available kernel threads on the GPU, then each kernel thread has $\alpha O(P)$ work where $\alpha > 1$. This means that the parallel implementation scales linearly with number of points, which is not desirable as the number of points increases.

- Global Memory Access: The input vertices V are stored in global memory and are shared between all kernel threads during computation. This may prevent kernel threads from retaining some vertices in its L2 cache. Better memory arrangements can be made to move sections of the vertices into shared memory of each block and swap the sections between blocks after threads in each block complete their sections.

- Memory False Sharing: Each kernel thread writes single pixels to a global pixel map which may lead to memory false sharing over a single cache line on block boundaries as there are multiple pixels (of size `int2`) on a cache line.

3.3.2 Grid-Based Voronoi Generation

Because brute-force approach did not scale with respect to the number of points, we further explored a divide-and-conquer approach. Similar to the parallelization approach in assignment 2, we decompose the pixel map into grids and map each grid to a thread block on the GPU. For each pixel, the two phases of the algorithm are described below and a diagram outlining the approach is shown in Figure 7.

Phase 1 - Find Vertices in each Grid: The pixel map is divided into a 64×64 pixel grid and each grid is assigned to a CUDA thread block. Within each block, each thread is assigned a number of vertices and checks if each vertex is within the bound of its grid. If the vertex is within bounds, the vertex index is recorded in a shared memory array within the block and is later packed compactly using `SharedMemoryExclusiveScan`.

Phase 2 - Compute the Nearest Point for Pixel: After Phase 1, for each grid, we obtain a list of vertices within the grid. Within each thread block, we then map each kernel thread to 4 pixels in the grid. For each pixel, we determine the closest vertex to the pixel based on the vertices S in its grid and in the neighboring grids. If the set S is empty, we default to the brute-force approach where we scan all input vertices for the particular pixel.

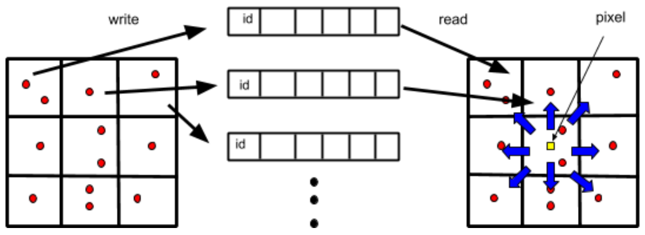
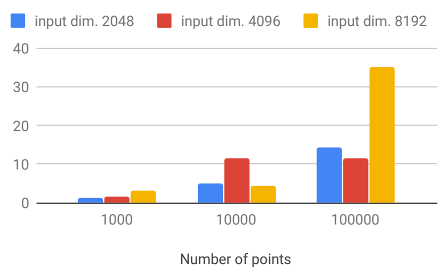


Figure 7: Illustration of the Grid-based Approach where the pixel map is decomposed into grids. Threads in each grid find the vertex ID inside the grid, and the IDs are used when computing the nearest vertex to a pixel.

This algorithm effectively partitions the pixel map and avoids extra work for each thread by restricting the number of vertices to check. Given a uniformly distributed set of vertices that are dense enough, the algorithm yields reasonable performance improvements. Indeed, as shown in Figure 8, compared with the brute-force approach, the Grid-Based Voronoi Generation approach yields considerable speedups across different input dimensions and number of points. Furthermore, Figure 8 shows that the percentage of time spent on Voronoi diagram generation (percentage of P1 + P2) is lower for the same workload when compared to the brute force approach. This indicates that the Grid-based approach is an improvement over the brute force approach.



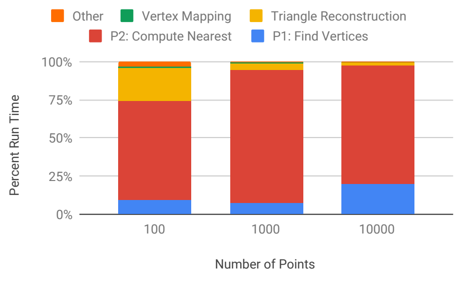
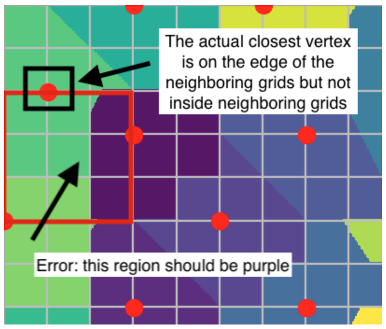


Figure 8: Speedup diagram and runtime distributions for the Grid-based approach. The speedup is computed with respect to the runtime of the brute force approach. The runtime distribution is obtained using uniformly distributed inputs with dimension of 1024.

However, with more detailed analysis, we observed several major limitations in this approach that affected its performance and overall correctness.

- Large Memory Requirement: Let the number of input points be P, the number of partitioned grids be n, and the input dimension be N. Phase 1 of this Grid-Based Voronoi approach requires at least $O(nP)$ memory. We note that $n \propto N$. This linear scaling of memory with respect to N and P is problematic, as the algorithm runs out of memory for inputs with $P > 1$ million. This severely limits the robustness of the algorithm.

- Grid-Based Voronoi Generation does not produces exact Discrete Voronoi diagrams: the approach is based on the key assumption that the vertex with the minimum distance to a pixel is only within the set of vertices in the neighboring grids of the pixel. However, this assumption does not always hold. An example of a failure case is shown in Figure 9. Although this limits robustness of the Voronoi diagram produced, we observe that the triangulation results produced in a majority of test cases are still Delaunay since the Triangle Reconstruction phase of the Triangulation algorithm only looks at the intersection of 3 or more vertex colors.



The actual closest vertex is on the edge of the neighboring grids but not inside neighboring grids
Error: this region should be purple

Figure 9: An example where the Grid-based approach fails to generate the correct Voronoi diagram

- Limited Cache Locality: The input vertices are stored in a global memory array in no particular order. For each pixel, the access of the vertices in neighboring grids occurs in almost random order. This causes a large number of cache misses when computing the closest vertices to each pixel. The input vertices can be sorted based on x and y coordinate values to improve locality.

- Load Balancing Issues: The runtime of this implementation depends heavily on the distribution of the input vertices. In particular, if the input vertices are clustered in certain areas and scattered in others, the algorithm will perform poorly as the thread blocks mapped to the vertex cluster will have less work than the threads mapped to the other areas. An analysis on the effects of input distribution on this approach is outlined in the Results section.

3.3.3 Jump Flood Voronoi Generation

Although the Grid-Based Voronoi Generation approach presented significant limitations in performance and correctness, it provided insights in the effectiveness of geometric-decomposition in GPU parallelization of Voronoi diagrams. After much literature survey, we discovered an adaptation of the Jump Flood Algorithm to generate Voronoi Diagrams on the GPU proposed by Rong and Tan [10] which produced much better speedup performances than other approaches.

The Jump Flood Algorithm works in rounds where each pixel checks the 8 neighboring pixels that are of `step_size` away from the pixel. The pixel then updates its closest vertex based on the coordinates of the closest vertices of its neighboring pixels. The pseudocode for the algorithm is shown below:

Algorithm 1 Jump Flood for Voronoi Diagram Generation

```
$V \leftarrow$ list of input vertices
$N \leftarrow$ input dimension
$B[2] \leftarrow$ double buffer each of size $N \times N$
$Bindex \leftarrow 0$ Copy all coordinates in $V$ to $B[Bindex]$
$step\_size \leftarrow N$
**while** $step\_size \geq 1$ **do**
    **for** each pixel $p$ in buffer $B[Bindex]$ **do**
        Obtain closest vertex coordinates of the 8 neighbor pixels of $p$ that are $step\_size$ away from $B[Bindex]$.
        Update the closest vertex to $p$ in $B[1 - Bindex]$
    **end for**
    $Bindex \leftarrow 1 - Bindex$
    $step\_size \leftarrow step\_size/2$
**end while**
```

We implemented the Jump Flood Algorithm in CUDA based on the implementation available by Rong et al. [7]. Specifically, we parallelized the inner for-loop of the algorithm using the GPU by mapping each kernel thread to a pixel in the pixel map and synchronizing all kernel threads at the end of each while loop iteration. We also used a double buffer twice the size of the pixel map to store the coordinates of the closest vertex for each pixel.

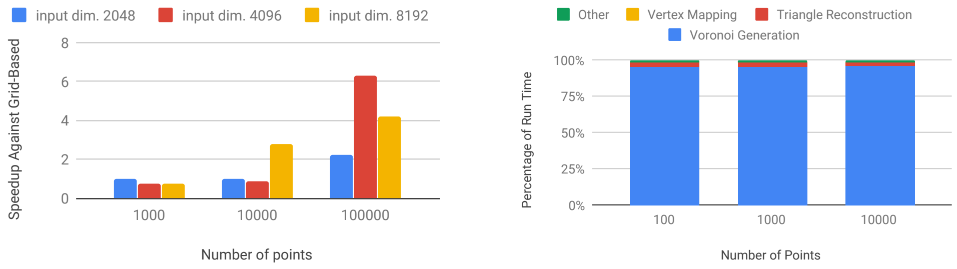


Figure 10: Speedup diagram and runtime distributions for the Jump Flood approach. The speedup is computed with respect to the runtime of the Grid-based approach. The runtime distribution is obtained using uniformly distributed inputs with dimension of 1024.

From Figure 10, we observe that the Jump Flood Algorithm yields significantly better speedup performances when compared to the Grid-based approach as the number of points increases. In instances with large number of input vertices or large input dimensions, the Jump Flood algorithm produces a 6$\times$ speedup compared to the Grid-based approach. This is the best performing approach among all of our parallel implementations.

The efficiency of the Jump Flood Algorithm can be fundamentally attributed to the reduced workload on each thread. In particular, given an input dimension of N, each thread performs $O(\log N)$ work since the step size reduces by a factor of 2 in each round. Aside from pixels near the edge of the pixel map, the algorithm achieves good load balance between threads as each thread only checks its neighboring pixels. By using a double buffer, we can perform asynchronous updates where we read pixel values from one buffer and write new pixel values to another buffer before swapping the buffers at the end of each round to achieve complete data-parallelism. There are no locks or atomic operations necessary in this implementation.

However, the Jump Flood Algorithm does suffer from relatively large memory requirements. Given an input dimension of N, the algorithm requires $O(2N^2)$ additional memory for the double buffer. This may be a limiting factor for vertices over large input dimensions.

3.3.4 Parallel Banding Algorithm

We looked at the Parallel Banding Algorithm (PBA) [11], a near state-of-the-art GPU approach for Discrete Voronoi diagram generation. Due to the implementation complexity of PBA, we did not implement this algorithm but studied an existing CUDA implementation by Rong et al. [7]. PBA further decomposes the pixel map geometrically and generally consists of the following 3 phases:

> **Phase 1 - Band Sweeping:** Compute 1D Voronoi diagrams for each row of the pixel map by parallelizing across bands of each column.
>
> **Phase 2 - Hierarchical Merging:** Compute proximate nearest vertices for each column of the pixel map using the 1D Voronoi diagram
>
> **Phase 3 - Block Coloring:** Use the proximate nearest vertices for each column to compute the nearest vertex for each pixel in the column.

We compared the performance of our Voronoi generation approaches with the existing PBA implementation and presented the respective speedups in our Results section below.

4 Results

4.1 Correctness Validation

We wrote a script to validate the resulting Delaunay Triangulation. The script performs the following three tests:

1. Each input point is a vertex of one or more triangles.
2. For a given triangle, the vertices of the triangle are all different.
3. No point is inside the circumcircle defined by a triangle.

To perform the third test, we had a pre-processing step where we ensured that the vertices were in counterclockwise order. For vertices $v_1 = (x_1, y_1), v_2 = (x_2, y_2), v_3 = (x_2, y_3)$ and all points $p_i = (x_i, y_i) \in V$ s.t. $p_i \notin \{v_1, v_2, v_3\}$, we took the determinant $det = \begin{bmatrix} x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \\ x_i & y_i & x_i^2 + y_i^2 & 1 \end{bmatrix}$.

If $det < 0$, then the point p_i is outside the circle. If $det = 0$, then the point is on the circle. If $det > 0$, then the point is inside the circle, violating the Delaunay condition. All outputs have been tested using this script.

4.2 Testing Environment

We ran the various GPU implementations on an Nvidia Tesla K80 GPU with CUDA 9.0. We ran the CPU sequential baseline on a 3.20GHz Intel Xeon CPU E5-1660. We used the C library function `clock` for measuring the runtime of each implementation.

4.3 Implementation Performance

We measured the performance of each parallelization approach based on speedup over the single-threaded CPU baseline implementation obtained from the PBBS package [5]. Specifically, we vary our test cases by the number of input points, the input dimensions, and the input distributions and observe the speedup behaviors at each case. The speedup graphs of each approach under different inputs are plotted in Figures 11-17.

From the results, we first observe that while the Brute-force and Grid-based approaches did not achieve any speedup over the baseline implementation, the Jump Flood and PBA approach did achieve a significant speedup over the baseline for cases with a large number of input points and large input dimensions (with the speedup reaching a maximum of $118\times$). Note that in the case of input dimension of 2048×2048 for 10K and 100K points, the CPU implementation is so fast that it registered an execution time of near 0. Table 1 tabulates the runtime of each GPU implementations under the different inputs.

CPU Implementation vs. GPU Implementation: It is interesting to note that while speedup is achieved on workloads with large number of inputs, for smaller input sizes such as 100 or 1000 points, no GPU implementations measured were able to achieve a speedup of > 1 over the CPU implementation. We hypothesize that this interesting observation can be attributed to fundamental differences between CPU and GPU implementations. In particular, a larger percentage of the runtime is attributed to the overhead for the kernel launches and GPU memory operations for GPU programs when dealing with less computationally heavy workloads. In these lighter workloads, CPU implementations perform better since they require less setup and launch overhead. On the other hand, when dealing with computationally heavy workloads, GPU implementations achieve significant speedups over the CPU implementations since the percentage of time spent on kernel setup and host-to-device or device-to-host memory transfer diminishes compared to the percentage of time spent performing the computation.

4.3.1 Effect of Varying Input Dimension on Performance

Figures 11-14 show the speedup for different input dimensions while holding the number of input points constant.

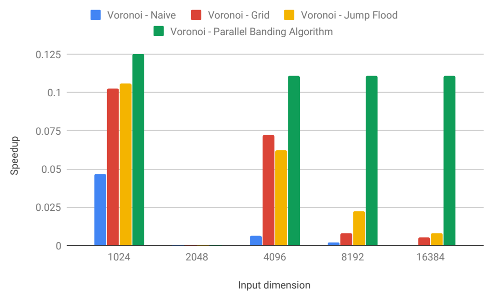


Figure 11: Input dimension vs. speedup for 10K points

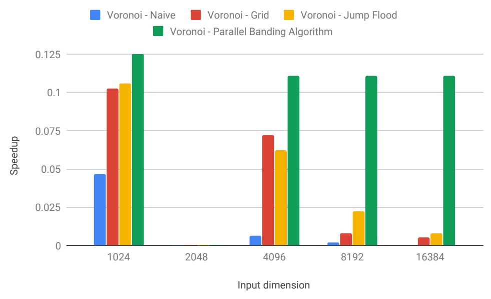


Figure 12: Input dimension vs. speedup for 100K points

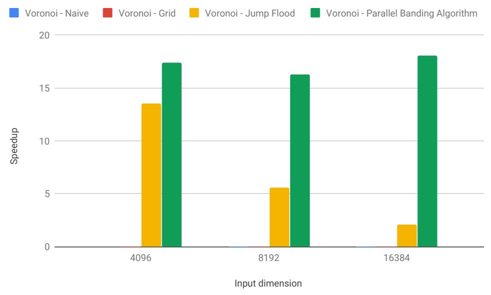


Figure 13: Input dimension vs. speedup for 1M points

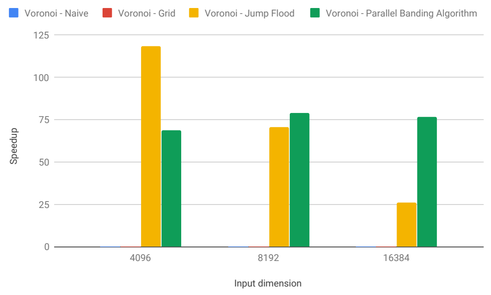


Figure 14: Input dimension vs. speedup for 10M points

From the above result, we observe that both the Brute-Force Voronoi and Grid-based Voronoi approaches failed to achieve any speedup over the baseline as input dimensions increase. Specifically, the runtime of the brute-force approach increased quadratically with the increase in input dimensions as shown by the runtimes in Table 1.

We also observe that the performance for Jump Flood Algorithm decreases as the input dimension increases. This suggests that the Jump Flood Algorithm does not scale well with increased input grid dimensions. In fact, the runtime of the Jump Flood Algorithm is highly dependent on the input dimension. This is because a larger input dimension leads to a larger initial step size and therefore more iterations of the algorithm, as the step size decreases by a factor of 2.

Compared to the Jump Flood Algorithm, the PBA implementation scales well with increased input dimensions as the speedup does not vary significantly across different grid sizes. This indicates that the fine-grained geometric decomposition employed by PBA is more effective in dealing with larger grid sizes.

4.3.2 Effect of Varying Number of Points on Performance

Figures 15-17 show the speedup for different numbers of points while holding input dimension constant.

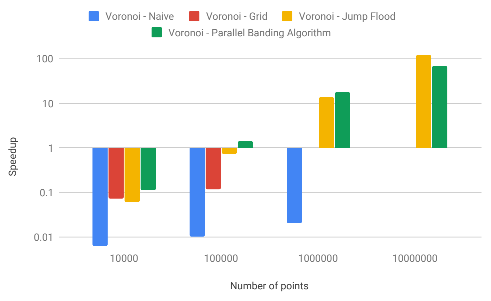


Figure 15: Number of points vs. speedup for input dimension 4096

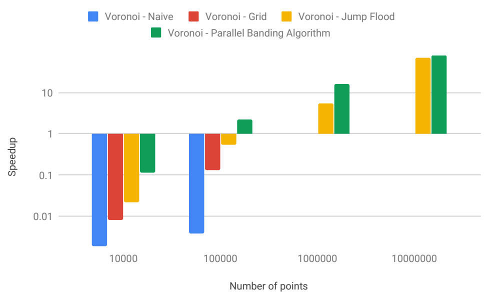


Figure 16: Number of points vs. speedup for input dimension 8192

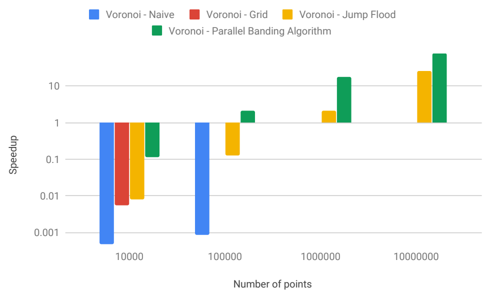


Figure 17: Number of points vs. speedup for input dimension 16384

From the above results, we again observe that Brute-Force Voronoi and Grid-based Voronoi approaches failed to achieve any speedup over the baseline. The runtime of the Brute-force approach again increased quadratically with respect to increasing data points.

We also observe that the speedup for Jump Flood algorithm increases with increased number of input points. From the runtimes shown in Table 1, we observe that this is not because the Jump Flood implementation runs faster with increased number of points, but because the baseline runs slower with large input sizes. We also observe that the changes in run time for the Jump Flood algorithm across increasing number of input points is fairly small, indicating that the Jump Flood algorithm scales well with increased number of input points. This can be explained by the fact that the complexity of the Jump Flood algorithm is independent of the number of input points and only dependent on the input dimension. We notice that the Jump Flood approach yields comparable performance to the PBA approach with varying input points.

4.3.3 Effect of Different Input Distributions on Performance

We observe that for Brute-force, Jump Flood, and PBA, varying the input distribution while keeping input dimensions and number of input points constant does not change performance. However, as mentioned in the above section, the Grid-based implementation is highly dependent on the distribution of input points. As shown in Figure 18, we observe that the Grid-based algorithm runs slower on the same input dimensions and same number of points generated on Gaussian distributions than Uniform distributions. This is due to the fact that there are more pixels with no neighboring vertices in the Gaussian distribution, leading to an increased number of brute-force pixel colorings.

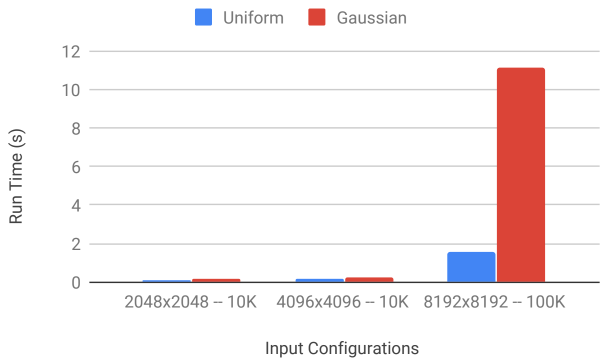


Figure 18: Uniform vs. Gaussian distributions for the Grid algorithm

4.3.4 Performance Breakdown

The execution time for the Jump Flood Algorithm is decomposed in Figure 19 with inputs containing 10M points and varying input dimensions. From the plot below, we notice that the performance bottleneck still lies with the Voronoi Generation Phase (indicated by Jump Flood in the diagram). However, compared to the execution time breakdown for other approaches, the Jump Flood algorithm demonstrates better scaling behavior as the percentage of time spent on Voronoi diagram generation is lower than Brute-Force and Grid-based approaches.

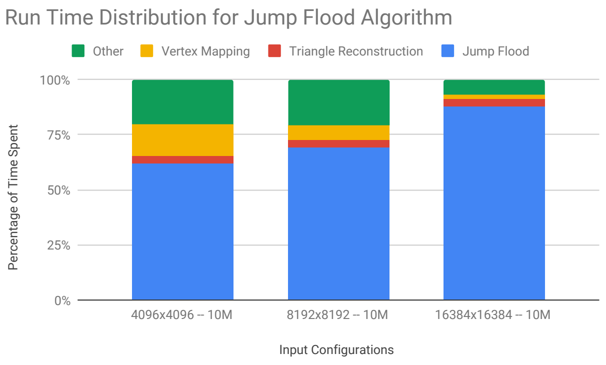


Figure 19: Breakdown of Execution Time for the Jump Flood Algorithm

4.4 Implementation Bottleneck

We used the Nvidia profiler (`nvprof`) to examine our best-performing implementation, the Jump Flood algorithm, to see what is limiting speedup. The kernel that performed jump flood was responsible for >90% of the total runtime. The profiler indicated that the warps and threads were fully utilized, so occupancy is not the issue.

Looking at the compute and memory utilization of the kernel in Figure 20, we see that both compute throughput and memory bandwidth are below 40%, suggesting that the issue is memory latency. Figure 21 shows the reasons for stalls. We see that execution dependencies are the primary reason for stalls, as input required by some instruction is not yet available.

To improve the algorithm, we can make use of shared memory. Currently, the kernel is only using global memory, which has higher latency and lower bandwidth than shared memory. Jump flood does not access nearby locations, but jumps to other locations (random accesses of the cache), so we will have to think more about how shared memory could be used.

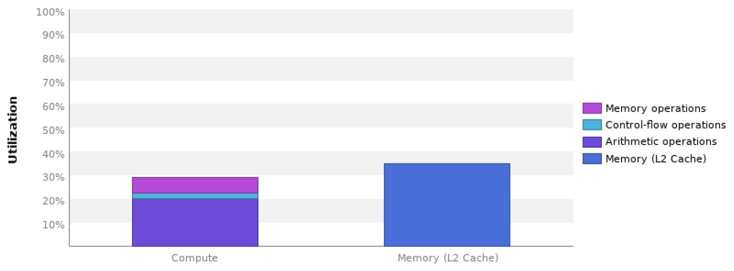


Figure 20: Compute and memory utilization of the Jump Flood kernel

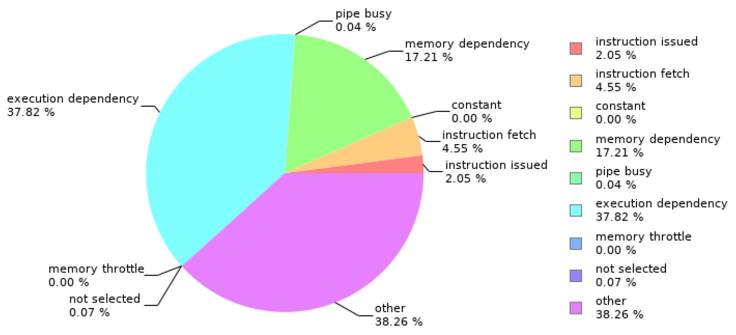


Figure 21: Reasons for stalls in the Jump Flood kernel

5 Conclusion

In this project, we attempted to parallelize algorithms for 2D Delaunay Triangulation using the GPU. We utilized the property that the Voronoi diagram is the dual for Delaunay Triangulation and implemented the Brute-Force, Grid-based, and Jump Flood GPU parallelization approaches. Using the single-threaded CPU sequential implementation in PBBS [5] as baseline, we observe that the Brute-Force and Grid-based approaches fail to yield speedups while the Jump Flood implementation yielded comparable speedups to the PBA implementation with increasing number of input points. These results produced insights in how improvements in the runtime and work distribution of parallel algorithms can significantly improve the performance of the parallel implementation.

6 Work Distribution

The work is distributed equally 50%-50% among the two group members. We distributed the work as follows: Harvey implemented the grid Voronoi and Jump Flood Voronoi, and a Voronoi visualizer. Connie set up the environment, code scaffold, and Makefile while also completing the brute force Voronoi implementation and profiling the PBBS parallelized implementation. Both wrote scripts to automate the experiments, analyzed data, and wrote this report.

References

[1] Pranav Kant Gaur and SK Bose. On recent advances in 2d constrained delaunay triangulation algorithms. *arXiv preprint arXiv:1707.05949*, 2017.

[2] Valentin Fuetterling, Carsten Lojewski, and Franz-Josef Pfreundt. High-performance delaunay triangulation for many-core computers. In *High Performance Graphics*, pages 97–104, 2014.

[3] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 467–478. ACM, 2016.

[4] A demo of delaunay triangulations and voronoi diagrams, Dec 2009. URL `https://wildabc.wordpress.com/2009/12/11/a-demo-of-delaunay-triangulations-and-voronoi-diagrams`.

[5] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM, 2012.

[6] Jean-Daniel Boissonnat and Monique Teillaud. On the randomized construction of the delaunay tree. *Theoretical Computer Science*, 112(2):339–354, 1993.

[7] G.D. Rong, T.S. Tan, Thanh-Tung Cao, and Stephanus. Computing two-dimensional delaunay triangulation using graphics hardware. In *The 2008 ACM Symposium on Interactive 3D Graphics and Games*, pages 89–97, 2008. URL `http://www.comp.nus.edu.sg/~tants/delaunay.html`.

[8] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2d constrained delaunay triangulation using the gpu. *IEEE transactions on visualization and computer graphics*, 19(5):736–748, 2013.

[9] Dan Maljovec. Delaunay triangulation on the gpu, 2010. URL `http://www.cs.utah.edu/~maljovec/files/DT_on_the_GPU_Print.pdf`.

[10] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116. ACM, 2006.

[11] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90. ACM, 2010.

7 Appendix

Input Dimension	Number of Points	Serial CPU Random Incremental (Baseline)	Voronoi Naive (Ours)	Voronoi Grid (Ours)	Voronoi Jump Flood (Ours)	Voronoi Parallel Banding Algorithm[7]
1024	100	0	0.0888	0.0879	0.0933	0.08
1024	1000	0	0.1014	0.0913	0.0928	0.09
1024	10000	0.01	0.2133	0.0976	0.0946	0.08
2048	1000	0	0.1369	0.1055	0.1088	0.08
2048	10000	0	0.5545	0.1075	0.1104	0.09
2048	100000	0.01	3.672	0.2541	0.114	0.09
4096	1000	0	0.2055	0.1344	0.1802	0.09
4096	10000	0.01	1.6038	0.1389	0.1602	0.09
4096	100000	0.14	13.3908	1.1754	0.1875	0.1
4096	1000000	2.78	135.7671	fail	0.206	0.16
4096	10000000	40.63	200+	fail	0.3432	0.3469
8192	1000	0	0.7769	0.2453	0.3385	0.08
8192	10000	0.01	5.4726	1.2678	0.4515	0.09
8192	100000	0.2	53.5301	1.5291	0.3644	0.09
8192	1000000	2.77	200+	fail	0.4998	0.17
8192	10000000	45.04	200+	fail	0.6375	0.57
16384	1000	0	2.5158	0.8165	1.3304	0.09
16384	10000	0.01	21.5332	1.8596	1.2619	0.09
16384	100000	0.19	217.7283	fail	1.4874	0.09
16384	1000000	2.89	200+	fail	1.3849	0.16
16384	10000000	44.32	200+	fail	1.7175	0.58

Table 1: Total clock time (seconds) for various approaches.