High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

High-Performance Delaunay Triangulation for Many-Core Computers

V. Fuetterling and C. Lojewski and F.-J. Pfreundt

Fraunhofer ITWM, Germany

Abstract

We present an efficient implementation of a Dwyer-style Delaunay triangulation algorithm that runs in $O(N)$ expected time. An implicit quad-tree is constructed directly from the floating point bit patterns of the input points by sorting the corresponding Morton codes with a radix sorting procedure. This unique structure adapts elegantly to any (non-)uniform distribution of input points and increases the accuracy of the merging calculations by grouping floating point values with similar bit patterns. Our implementation allows for easy parallelization and we demonstrate a record construction speed of one Billion Delaunay triangles in just 8s on a many-core SMP machine.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms

1. Introduction

The Delaunay Triangulation (DT) is a fundamental method in the field of computational geometry. Applications include point location, path finding and visual computing, in particular image processing and mesh generation. Various algorithms in three categories have been proposed to compute the DT efficiently. The Sweepline algorithm has been invented by [For86], incremental algorithms have been proposed by [GKS90], [ACR03], [Buc09] and divide-and-conquer algorithms are due to [GS85], [Dwy87], [She96b]. All these algorithms were designed with sequential execution in mind. Since their introduction computer hardware has evolved, but publications concerned with the adaption of the dated algorithms are sparse, at least on the CPU. On the GPU multiple parallel algorithms have been proposed by [RTCS08], [QCT12] and [CNGT14] that have successively increased DT construction speed well beyond the publicly available counterparts on the CPU such as CGAL [CGA14] and *Triangle* [She96b]. Our goal is to adapt one of the aforementioned sequential algorithms to the parallel capabilities of modern CPUs. We chose the Dwyer style divide-and-conquer algorithm as it is considered the 'strongest overall' of the sequential algorithms [SD95] and is very well suited for task-parallelism. Furthermore this algorithm has been proven to have a time complexity of $O(N)$ for the uniform distribution of N coordinates under specific conditions [KK87]. We verify experimentally that our algorithm exhibits the same time complexity for various point distributions.

Our key contribution is the linear floating point quad-tree, an implicit data structure that allows our DT implementation to scale up to 32 CPU threads and beyond. The result is a highly efficient CPU implementation that substantially outperforms the GPU alternatives proposed by [QCT12] and [CNGT14]. In the following section we describe the floating point quad-tree data structure in detail. The implementation of the DT is topic of section 3 and in section 4 we present our experimental results and an exhaustive analysis. Section 5 concludes the paper with a summary of the key points and a proposal for future work.

2. The linear floating point quad-tree

The linear floating point quad-tree is a data structure based on Morton codes, which are derived from the binary representation of floating point values. We assume the representation to conform with the IEEE 754 standard.

2.1. Morton codes

Morton codes are indices along the space-filling z-order curve. The z-order curve maps n-dimensional coordinates (here $n = 2$) to a scalar value, the Morton codes. The mapping is performed in such a way that the resulting order of Morton codes is equivalent to a depth-first traversal of a particular quad-tree constructed from the coordinates. A Mor-

DOI: 10.2312/hpg.20141098

ton code is computed by interleaving the bit patterns of the components of a coordinate. This representation of a quad-tree was referred to as 'linear quad-tree' by [Gar82] and successfully applied by [LGS*09] to rapidly generate bounding volume hierarchies (BVH) for triangular meshes.
As an alternative to the z-order curve the related Hilbert curve can also be used to construct the same quad-tree. However, Morton codes are faster to compute and conceptually simpler.

2.2. Construction

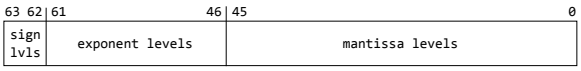
A linear quad-tree is constructed by sorting the input coordinates by their respective Morton codes. Let $\mathcal{B}$ and $\mathcal{F}$ denote the binary value and the real value of a floating point value respectively. Since lexicographical order and numerical order differ for floating point values, applying the Morton codes directly to $\mathcal{B}$ presents a problem. A binary representation $\mathcal{B}'$ with the following property is required:

$$\mathcal{F}_i < \mathcal{F}_j \Leftrightarrow \mathcal{B}'_i < \mathcal{B}'_j$$

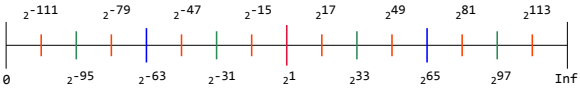
If $\mathcal{F}$ is restricted to positive values this requirement is actually fulfilled. Consequently for negative values the lexicographical order is inverse to the numerical order. Also, because of the sign bit, negative values are greater than positive values in lexicographic order.
Hence a suitable representation $\mathcal{B}'$ can be achieved by inverting the sign bit, exponent and mantissa for negative $\mathcal{F}_i$ and setting the sign bit to '1' for positive $\mathcal{F}_i$.
In summary three steps are necessary to construct the linear floating point quad-tree: First, the coordinate components are transformed from $\mathcal{B}$ to $\mathcal{B}'$. Second, the Morton codes are applied. Third, the Morton codes are sorted.

2.3. Geometric structure

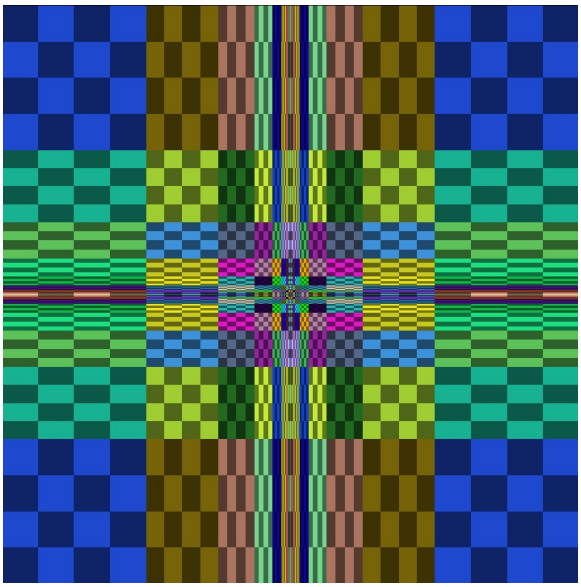
We want to explore the geometric structure of the floating point quad-tree, i.e. at what positions space is subdivided and at what *level* a particular subdivision occurs. The level reflects the hierarchical order of subdivisions and corresponds directly to a particular bit position in the Morton Code, e.g. the root node is at level 64 and its subdivision defined by the most significant bit (Figure 1a).
For simplicity we first consider a linear quad-tree defined by integer coordinates. With every level the space is subdivided exactly in half.The direction of subdivision is cyclic in X and Y (and Z etc. for higher dimensions). The resulting structure corresponds to a regular grid at every level of the linear quad-tree.
Floating point coordinates generate a different geometric structure due to the exponent bits. Their influence is best understood by studying Figure 1. Figure 1b depicts the hierarchical segmentation along a single axis and Figure 1c the subdivision of the four quadrant unit squares down to the least significant exponent level. The bits of the mantissa however are evenly spaced and form regular grids at every mantissa level. The base vectors of the regular grids have magnitudes that depends on the value of the exponent (see Figure 1c). The whole structure is symmetric with respect to the coordinate axes because the two most significant quad-tree levels correspond to the sign bits of the x and y components.



(a) Structure of the 64 bit Morton code.



(b) Subdivision of the first 4 exponent levels along a single axis. The hierarchical order of the subdivisions by color is: red → blue → green → orange.



(c) Subdivision of the four quadrant unit squares. Areas with different colors belong to different exponents. Alternating color shades within a single exponent area mark the regular grid formed by the mantissa levels.

Figure 1: Geometric structure of the floating point quad-tree.

2.4. Numeric structure

An interesting property of the floating point quad-tree is its rigorous numerical structure that sets it apart from other space partitioning schemes. We consider the coordinates grouped together in an area at level i. By construction, the bits from bit i to bit 63 are identical (Figure 1a). This property can be leveraged to increase the robustness of floating point calculations and to tighten error bounds. For example coordinates that share the same exponent can be subtracted without round-off error and the result will have a mantissa

with the $32 - i/2$ least significant bits set to zero. The unused bits can guarantee a subsequent exact addition or multiplication.

The construction of a Delaunay triangulation requires two arithmetic procedures, the *In-circle* and *Orientation* tests. The results of these predicates depend only on the sign of the calculation, but the sign needs to be *exact*. If the wrong sign is produced due to round-off errors, the DT is invalid and the construction algorithm may fail. Multi-precision floating point arithmetic is a reliable solution but also very slow. A sensible compromise performs error analysis on the inexact result and only employs Multi-precision arithmetic if the sign is questionable. Error analysis can be static, dynamic or a combination of both. Static analysis simply compares the magnitude of the result to a compile-time constant while dynamic analysis requires additional computations. Naturally dynamic analysis provides tighter error bounds whereas the quality of static analysis greatly depends on good preconditioning.

Preconditioning is the natural application for the numerical structure of the floating point quad-tree. Considering a random distribution of coordinates, minimal error bounds at every level (as a function of the exponent) can be computed at compile-time. Hence static error analysis can be very efficient especially in the lower levels of the quad-tree where most predicates are evaluated.

3. Implementation of the DT

The algorithm can be divided into three phases: Sort, Subdivision and Merge. Dedicated subsections 3.1, 3.2 and 3.3 describe the three phases in detail. The parallel implementation is the topic of subsection 3.4.

3.1. Phase 1: Sort

The sorting phase receives a list of input coordinates, transforms the floating point representation according to subsection 2.2 and encodes the coordinates with the Morton code. The Morton codes are then sorted with an efficient radix sort implementation, which is out-of-place, uses a key length of one byte and utilizes SIMD instructions and NUMA optimization to achieve a high memory bandwidth. After the sorting process the construction of the floating point quad-tree is concluded. Note that only the list of sorted Morton codes is passed on to subsequent phases since the mapping of Morton codes and coordinates is bijective.

3.2. Phase 2: Subdivision

The task of the subdivision phase is to descend down the floating point quad-tree hierarchy until an area contains only one or two coordinates. The triangulation of these quantities is trivial.

We use the triangular data structure of [She96b]: The convex hull of a triangulation is surrounded by dedicated hull triangles to facilitate the merge algorithm.

The result of the subdivision phase is a *partition*, a data structure that contains four pointers to the hull triangles attached to the minimum and maximum coordinates in x and y direction. A single coordinate has no hull triangle and hence is marked with a special value.

The subdivision phase is detailed in algorithm 1. The first conditional tests if only one coordinate remains in the input interval (line 1). The particular formulation of this condition allows to detect and gracefully handle degenerate input points (lines $2-3$). The second conditional checks for two remaining coordinates (line 4). If the number of coordinates is more than two (and coordinates are not degenerate), the input interval needs to be split at the next subdivision of the floating point quad-tree. The level of the next subdivision is determined by a binary xor of the first and the last Morton code in the in input interval and a subsequent BSR instruction [INT13] (lines $8-10$). This will yield the position of the most significant bit that is different, i.e. the level of the next subdivision. Then the input array is partitioned by finding the two adjacent Morton codes with a $'0'$ and a $'1'$ respectively at the subdivision level (lines $12-19$). Finally recursion is invoked (lines $20-21$) and the results are merged (line 22).

Algorithm 1 Subdivision of the floating point quad-tree.

Subdivide($lidx$, $ridx$)

```
 1: if points[lidx] is equal to points[ridx − 1] then
 2:     Decode points lidx to ridx − 1
 3:     return Partition with single point lidx
 4: else if ridx − lidx is equal to 2 then
 5:     Decode points lidx and lidx + 1
 6:     return Partition with points lidx and lidx + 1
 7: else
 8:     l ← lidx
 9:     r ← ridx − 1
10:     level ← BSR( points[l] ⊕ points[r] )
11:     mask ← Shift left 1 by level
12:     while not (points[l + 1] & mask) do
13:         m ← (l + r)/2
14:         if points[m] & mask then
15:             r ← m
16:         else
17:             l ← m
18:         end if
19:     end while
20:     left ← Subdivide(lidx, l + 1)
21:     right ← Subdivide(l + 1, ridx)
22:     return Merge(left, right)
23: end if
```

3.3. Phase 3: Merge

The merge phase produces a single Delaunay-triangulated output partition from two input partitions. The basic algo-

rithm for the merge phase is adapted from [GS85]. During the merge phase *Orientation* and *In-circle* predicates are evaluated repeatedly and the numerical structure of the floating point quad-tree can be leveraged to precompute tight error bounds. We employ error analysis in three stages. Stage 1 is similar to the 'Adaptive A' computation from [She96a] except for the error bounds. Stage 2 utilizes interval arithmetic to compute a minimal dynamic error bound. If stage 2 cannot decide the predicate, stage 3 computes the exact result with the aid of the GNU multi-precision library [GMP14].

3.4. Parallelism

Parallelism is inherent in a divide-and-conquer algorithm as every subdivision produces two independent partitions. However the scalability with many threads is often limited. The reason for this is that only one partition exists in the beginning and the number of independent partitions only doubles with every subdivision level. As a result, a maximum of 2^n threads can be active at level n. The same limitation applies to the final merging computations. If a significant amount of work must be done in the upper subdivision levels, Amdahl's Law prevents good scaling behavior for many threads. Hence the challenge for a scalable divide-and-conquer algorithm is to defer as much work as possible to the lower subdivision levels where all threads can participate.

Fortunately, due to the linear floating point quad-tree, subdivision of the upper levels is very fast with our algorithm. No data movement is required and the complexity of a subdivision is only $O(\log n)$ where n is the number of points in the current partition. Hence our implementation uses a single thread to quickly partition the floating point quad-tree into independent sub-trees (*bins*) with an appropriate number of coordinates. The subdivisions (or *splits*) are recorded in a *split-list* for the parallel merge phase (Figure 2). Every quad-tree level has a dedicated *split-list* and an atomic counter. Every *split* entry in a *split-list* has storage for the two input partitions of its subdivision and a pointer to where the merge result must be stored.

As soon as the master thread has finished the partitioning of the bins the algorithm proceeds as described in Figure 2. Note that no global synchronization is required during the entire triangulation procedure.

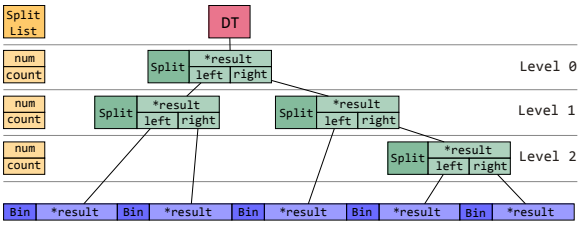


Figure 2: Data structure for the parallel merge phase. First the *bins* (sub-trees) are processed in parallel and the results are propagated to the corresponding *splits* as indicated by the black lines. Afterwards the *split-list* is traversed from the lowest to the highest level. The *num* field indicates the total number of *splits* at a particular level and *count* is an atomic counter used by the threads to acquire the next available *split*. The *left* and *right* fields of the *split* are tested and each thread will spin-wait in case the dependent results have not completed. Once a thread receives a *count* value larger than *num* it proceeds to the next *split-list* level. This procedure is repeated until the final partition has been finished on the highest level and the DT is complete.

4. Experimental Results

In order to benchmark our implementation (fqDel) we compare it to the two popular DT implementations *Triangle* and CGAL (version 4.2). *Triangle* is a Dwyer-style divide-and-conquer algorithm and thus quite similar to our own DT algorithm with the distinct difference of the linear floating point quad-tree (LFQT), and multi-threading support. Hence we can directly demonstrate the impact of our contributions. CGAL uses incremental point insertion with presorting the input coordinates along the Hilbert curve for speed. It is the fastest publicly available DT implementation known to the authors. The experimental setup consists of a dual-processor Intel Xeon E5-2670 system with 64 GB DDR3 Ram. The timings we report are derived by counting CPU cycles and normalizing the result to a frequency of 3 GHZ (the highest turbo boost frequency of the E5-2670 at full load). This ensures that the true scaling behavior of our algorithm is not masked by turbo states. Furthermore we repeat every measurement multiple times and report the minimum time in order to minimize OS interference. We also compare our work to GPU-DT [QCT12] and gDel2D [CNGT14], recent publications on GPU accelerated computation of the DT.

We have chosen five different point distributions generated randomly with a fixed seed for every measurement. The Uniform and Grid distributions approximate common real-world data, the Cluster is a common feature within real-world data, the Circle a common benchmark and the Spiral distribution has been chosen for its interesting geometry. Circle and Spiral are centered at the origin, whereas all other distributions are located in the first quadrant. The Grid is square and consists of integer coordinates only.

4.1. Single-threaded comparison

Figure 3 depicts the run-times of fqDel, CGAL and *Triangle* for multiple point distributions with input sizes of $100k$, $1M$, $10M$ and $100M$ points. fqDel is able to complete the DT much faster than either CGAL or *Triangle*, irrespective of point distribution and input size. Focusing on the $100k$ diagram, fqDel outperforms CGAL and *Triangle* by a factor of 2.5× (Uniform) to 3× (Spiral). The discrepancy for the Grid distribution is even higher. Neighboring grid points

are cocircular so that CGAL and *Triangle* have to resort to exact arithmetic while fqDel can decide the cocircular cases of integer grid coordinates by interval arithmetic. For larger input sizes the run-time ratio of CGAL and fqDel changes only slightly in favor of fqDel and the overall scaling appears linear. *Triangle* clearly does not scale linearly with increasing input size and as a consequence runs much slower than either fqDel or CGAL. *Triangle* failed to compute the DT for $100M$ input points. The asymptotic

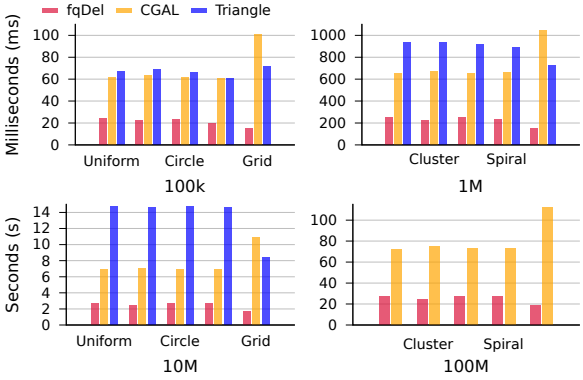


Figure 3: Single-threaded run-times of fqDel, CGAL and *Triangle* for multiple distributions with $100k$, $1M$, $10M$ and $100M$ points.

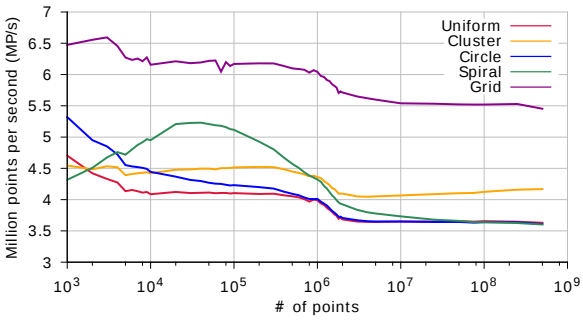


Figure 4: Asymptotic behavior of fqDel over a range from $1k$ to $500M$ points for multiple distributions. MP/s is the rate at which input points are processed (for a single thread). A constant rate corresponds to linear scaling.

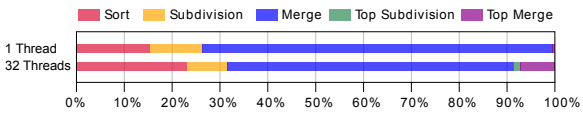


Figure 5: Timing break-down for the different phases of fqDel, for single-threaded and multi-threaded execution. The timings for *Subdivide* and *Merge* are reported separately for the lower and the upper (Top) part of the LFQT.

behavior of fqDel is detailed in Figure 4. The unit on the y-axis is the rate at which input points are processed, i.e. a rate independent of the input size is expected for linear scaling. We first concentrate on the Uniform curve which is mostly parallel to the x-axis with two significant exceptions. There is a first decline of the rate between 10^3 and 10^4 and a second around 10^6. These effects can be attributed to the influence of the L1/L2 and L3 caches respectively. If the largest part of the data set fits into one of the caches, memory access latency is significantly improved and hence the rate is higher. Disregarding the influence of the caches the rate is constant over a range of more than 6 orders of magnitude thus demonstrating an algorithmic complexity of $O(N)$. The same applies to the Cluster and Grid distribution albeit with a constant offset compared to the Uniform rate. The slight increase of the cluster rate towards huge input sizes is an artifact of a high number of degenerate points that are efficiently filtered out during triangulation (Section 3.2). The different rates for Uniform, Grid and Cluster are easily explained: A DT of a subset of the input points often contains many triangles that are not part of the DT of the complete set and hence must be deleted or restructured during the Merge phase (Section 3.3). The construction of invalid triangles is highly sensitive to the merge order of the subsets which, in our case, is dictated by the geometric structure of the LFQT. Hence the linear floating point quad-tree appears to be better suited for non-uniform point distributions such as the Cluster, Circle and Spiral, the latter two approaching the Uniform case for large input sizes. Even though the Grid is uniform, due to its regular structure triangles of a subset are also very likely triangles of the final DT. Thus the Grid has the highest rate of all the distributions.

Figure 5 shows the total run-time of fqDel broken down into the different phases in terms of percentages. The *Subdivide* and *Merge* phases are further separated into the time spent inside the lower part of the LFQT (bins) and the upper part of the LFQT (initial subdivision and parallel merge phase). For single-threaded execution the time spent in the upper part is almost non-existent as the number of bins is coupled to the number of threads, whereas in the case of 32 threads the upper part accounts for about 10% of the total run-time and more threads will further increase the ratio. The initial subdivision in our implementation is always single-threaded, although for higher thread counts a parallel implementation will likely be worth the effort. While the lower part of the LFQT consists of completely independent tasks, the upper part must produce a single result and hence cannot scale linearly with thread count, ultimately limiting scalability of the complete algorithm.

Table 1 provides counts to asses the influence of the LFQT's arithmetic structure on error analysis. Considering that the total number of *Orientation* and *In-circle* tests ranges from $40M$ to $80M$ for $10M$ input points, the semi-static error analysis proves to be highly efficient for both fqDel and *Triangle*. The exception is Grid with many co-

	Uniform	Cluster	Circle	Spiral	Grid
Triangle					
Orientation	15	33453	10	353	700614
In-circle	134	113023	292	5878	11520820
fqDel - *const*					
Orientation	10	31100	8	417	0
In-circle	121	103814	316	10685	9991921
fqDel - *level*					
Orientation	5	13	6	79	0
In-circle	64	15144	252	10224	9991921

Table 1: Total number of predicates that are within error bounds for *Triangle* and fqDel at $10M$ points. The fqDel error bounds labeled *const* are identical with those of *Triangle*, the fqDel error bounds labeled *level* are level-dependent.

circular cases. Comparing the level-dependent error bounds with the *Triangle* bounds, undecided predicates can be reduced significantly in relative terms for most distributions. In absolute terms however it is clear that run-times are only marginally affected, e.g. in case of the Cluster the overall speedup is about 0.5%.

4.2. Multi-threaded performance

The scaling of fqDel with core count is depicted in Figure 6, with and without Hyperthreading enabled. 16 cores are able to compute the DT $13-14\times$ as fast as a single thread. Enabling Hyperthreading increases the factor by about 30% regardless of core count. A $16c/32t$ configuration results in a factor of close to $17\times$ for the Uniform, Circle and Spiral distributions. For the Cluster and Grid distributions more work is deferred to higher subdivision levels which affects scalability slightly.

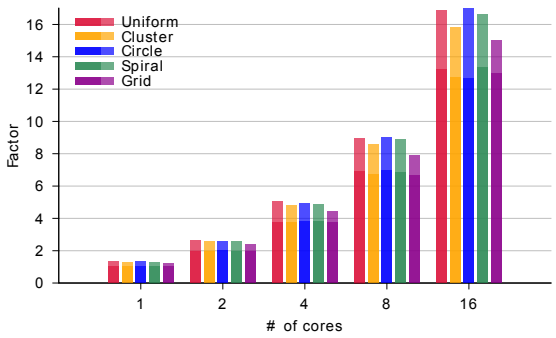


Figure 6: Scaling of fqDel with core count, illustrated for multiple distributions. Hyperthreading is enabled for the translucent part and disabled for the opaque part of the bars.

Triangle and CGAL do not offer multi-threading support. [BMPS09] has experimented with a multi-threading extension for CGAL, but the results demonstrated have limited scalability for the 3D DT, result for the 2D DT have not been published. Figure 7 compares fqDel to the latest advancements in GPU accelerated DT instead, namely GPU-DT [QCT12] and gDel2D [CNGT14] (both implemented in CUDA). The GPU results have been reconstructed from the respective publications and might be slightly inaccurate. Both results have been performed in double precision. According to [CNGT14] the DT on the GPU is memory bound so the results are still comparable in our opinion (the memory footprint of the DT should be increased by 15% at most). As depicted by Figure 7 fqDel is more than $4\times$ and $8\times$ faster than GPU-DT and gDel2D respectively on the NVIDIA Geforce GTX 580. [CNGT14] reports a general 30% speedup of the NVIDIA Geforce Titan over the GTX 580 for the DT in 3D. We have included this result as a reference in Figure 7, but we have to point out that it was not actually measured.

Comparing the radix sort speed of CPU and GPU, our implementation is able to outperform the Thrust-based implementation provided by the CUDA 6 SDK (running on a NVIDIA Titan) by a factor of 3.9, requiring only $0.89ms$ to sort $1M$ floating point values.

Even though the comparison of CPU and GPU results are not straightforward, considering the substantial lead in Figure 7 it is save to say that fqDel can compute the DT more efficient than any GPU algorithm published so far. The separate bar in Figure 7 illustrates the run-time for $500M$ uniform input points which result in $1G$ Delaunay triangles in just $8s$. This order of magnitude cannot be reached by GPUs due to memory limitations.

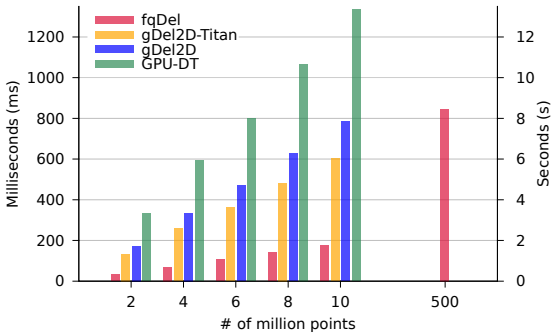


Figure 7: CPU vs. GPU. Run-times of fqDel, gGPU2D and GPU-DT for the uniform distribution with input sizes from $2M$ to $10M$ points. $500M$ points can be accomplished only by fqDel, GPUs do not offer a sufficient amount of memory.

4.3. Discussion

The experimental results prove that fqDel is a highly competitive DT implementation. Even though *Triangle* and fqDel are based on the same Dwyer-style divide-and-conquer algorithm fqDel is substantially faster even in the single-threaded case. The speedup must thus be attributed to the LFQT because otherwise the two implementations

are quite similar. The most notable improvements are multi-threading scalability (Figure 6) and asymptotic behavior. While *Triangle* has to resort to a $O(N \log N)$ Quicksort-based pre-sort procedure, fqDel can employ a fast $O(N)$ radix sort (Figure 3). Algorithmic complexity aside the floating point quad-tree offers some unique and interesting features. The structure of the tree can adapt to any distribution, no matter how sparse or dense or degenerate, while the uniform distribution appears to be the performance baseline (Figure 4). The numeric structure can minimize the requirement for exact arithmetic (Table. 1) and the bijection of input coordinates and Morton codes reduces the memory footprint.

A limitation of the proposed algorithm is that it does not expose data parallelism that could be mapped efficiently to the SIMD instructions of modern CPUs. Our implementation utilizes SIMD instructions only for the *Orientation* and *In-Circle* predicates for a total speed-up of 1% and 10% respectively.

An interesting question is if the LFQT is also compatible with point insertion algorithms (e.g. CGAL). Point insertion performance is highly sensitive to the ordering of the input points and benefits from the better locality preservation of the Hilbert curve compared to Morton order. As noted in Section 2.1 the LFQT can also be constructed from Hilbert order and in this case point insertion could be employed efficiently to triangulate the bins (Section 3.4) in parallel.

5. Concluding Remarks

In this paper we have introduced a novel data structure, the linear floating point quad-tree. Based on the LFQT we have demonstrated a highly efficient Dwyer-style divide-and-conquer DT implementation in 2*D* that exhibits excellent multi-threading scalability. To our knowledge the results are the fastest ever published, competing with established CPU solutions and CUDA based GPU implementations. In the future it would be interesting to extend our DT to higher dimensions and to answer the question if point insertion can be significantly faster than the divide-and-conquer approach in the lower levels of the LFQT constructed from Hilbert order. Furthermore we believe that the usefulness of the LFQT is not limited to the DT and should be considered when designing data structures for other applications.

References

[ACR03] AMENTA N., CHOI S., ROTE G.: Incremental constructions con brio. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry* (New York, NY, USA, 2003), SCG '03, ACM, pp. 211–219. URL: http://doi.acm.org/10.1145/777792.777824, doi:10.1145/777792.777824. 1

[BMPS09] BATISTA V. H., MILLMAN D. L., PION S., SINGLER J.: Parallel geometric algorithms for multi-core computers. In *Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry* (New York, NY, USA, 2009), SCG '09, ACM, pp. 217–226. URL: http://doi.acm.org/10.1145/1542362.1542404, doi:10.1145/1542362.1542404. 6

[Buc09] BUCHIN K.: Constructing delaunay triangulations along space-filling curves. In *Algorithms - ESA 2009*, Fiat A., Sanders P., (Eds.), vol. 5757 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 119–130. URL: http://dx.doi.org/10.1007/978-3-642-04128-0_11, doi:10.1007/978-3-642-04128-0_11. 1

[CGA14] CGAL: Computational geometry algorithms library. http://www.cgal.org, 2014. 1

[CNGT14] CAO T.-T., NANJAPPA A., GAO M., TAN T.-S.: A gpu accelerated algorithm for 3d delaunay triangulation. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2014), I3D '14, ACM, pp. 47–54. URL: http://doi.acm.org/10.1145/2556700.2556710, doi:10.1145/2556700.2556710. 1, 4, 6

[Dwy87] DWYER R. A.: A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica 2*, 1-4 (Nov 1987), 137–151. doi:10.1007/BF01840356. 1

[For86] FORTUNE S.: A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry* (New York, NY, USA, 1986), SCG '86, ACM, pp. 313–322. URL: http://doi.acm.org/10.1145/10515.10549, doi:10.1145/10515.10549. 1

[Gar82] GARGANTINI I.: An effective way to represent quadtrees. *Commun. ACM 25*, 12 (Dec. 1982), 905–910. URL: http://doi.acm.org/10.1145/358728.358741, doi:10.1145/358728.358741. 2

[GKS90] GUIBAS L. J., KNUTH D. E., SHARIR M.: Randomized incremental construction of delaunay and voronoi diagrams. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming* (New York, NY, USA, 1990), Springer-Verlag New York, Inc., pp. 414–431. URL: http://dl.acm.org/citation.cfm?id=90397.91347. 1

[GMP14] GMP: The GNU multiple precision arithmetic library. https://gmplib.org/, 2014. 4

[GS85] GUIBAS L., STOLFI J.: Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph. 4*, 2 (Apr. 1985), 74–123. URL: http://doi.acm.org/10.1145/282918.282923, doi:10.1145/282918.282923. 1, 4

[INT13] INTEL: *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2: instruction set reference, a-z ed., September 2013. 3

[KK87] KATAJAINEN J., KOPPINEN M.: Constructing delaunay triangulations by merging buckets in quadtree order. Unpublished manuscript, 1987. 1

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on GPUs. *Computer Graphics Forum 28*, 2 (Apr 2009), 375–384. URL: http://dx.doi.org/10.1111/j.1467-8659.2009.01377.x, doi:10.1111/j.1467-8659.2009.01377.x. 2

[QCT12] QI M., CAO T.-T., TAN T.-S.: Computing 2d constrained delaunay triangulation using the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 39–46. URL: http://doi.acm.org/10.1145/2159616.2159623, doi:10.1145/2159616.2159623. 1, 4, 6

[RTCS08] RONG G., TAN T.-S., CAO T.-T., STEPHANUS: Computing two-dimensional delaunay triangulation using graphics hardware. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2008), I3D '08, ACM, pp. 89–97. URL: http://doi.acm.org/10.1145/1342250.1342264, doi:10.1145/1342250.1342264. 1

[SD95] SU P., DRYSDALE R. L. S.: A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry* (New York, NY, USA, 1995), SCG '95, ACM, pp. 61–70. URL: http://doi.acm.org/10.1145/220279.220286, doi:10.1145/220279.220286. 1

[She96a] SHEWCHUK J. R.: Robust adaptive floating-point geometric predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry* (New York, NY, USA, 1996), SCG '96, ACM, pp. 141–150. URL: http://doi.acm.org/10.1145/237218.237337, doi:10.1145/237218.237337. 4

[She96b] SHEWCHUK J. R.: Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Selected Papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering* (London, UK, UK, 1996), FCRC '96/WACG '96, Springer-Verlag, pp. 203–222. URL: http://dl.acm.org/citation.cfm?id=645908.673287. 1, 3