Accelerating Force-Directed Graph Drawing with RT Cores

Stefan Zellmann* University of Cologne
Martin Weier† Hochschule Bonn-Rhein-Sieg
Ingo Wald‡ NVIDIA

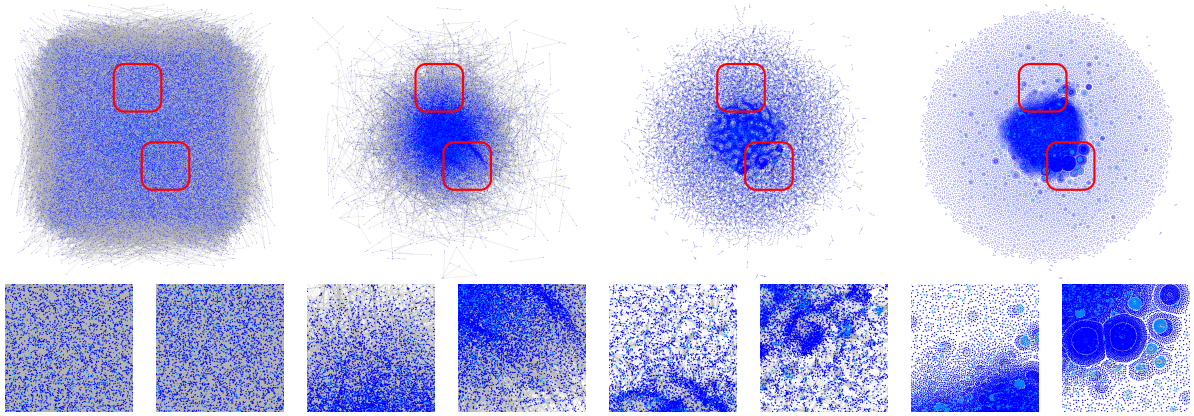

Figure 1: Drawing a Twitter feed graph (68K vertices, 101K edges) with a force-directed algorithm using RT cores. The images show the results after $N = 1$ (left), $N = 100$ (second from left), $N = 2{,}000$ (second from right), and $N = 12{,}000$ (right) iterations. We can generate these layouts in 0.003, 0.43, 7.35, and 39.3 seconds, and outperform a typical CUDA software implementation by 10.2×, 7.44×, 9.6×, and 10.9×, respectively.

ABSTRACT

Graph drawing with spring embedders employs a $V \times V$ computation phase over the graph's vertex set to compute repulsive forces. Here, the efficacy of forces diminishes with distance: a vertex can effectively only influence other vertices in a certain radius around its position. Therefore, the algorithm lends itself to an implementation using search data structures to reduce the runtime complexity. NVIDIA RT cores implement hierarchical tree traversal in hardware. We show how to map the problem of finding graph layouts with force-directed methods to a ray tracing problem that can subsequently be implemented with dedicated ray tracing hardware. With that, we observe speedups of 4× to 13× over a CUDA software implementation.

Index Terms: Human-centered computing—Visualization—Visualization techniques—Graph drawings; Computing methodologies—Computer graphics—Rendering—Ray tracing;

1 INTRODUCTION

Graph drawing is concerned with finding layouts for graphs and networks while adhering to particular aesthetic criteria [7, 32]. These can, for example, be minimal edge crossings, grouping by connected components or clusters, and obtaining a uniform edge length. Force-directed algorithms [8, 23] associate forces with the vertices and edges and iteratively apply those to the layout until equilibrium is reached and the layout becomes stationary.

Spring embedders, as one representative of force-directed algorithms, iteratively apply repulsive and attractive forces to the graph layout. The repulsive force computation phase requires $O(|V|^2)$ time over the graph's vertex set V. This phase can be optimized using data structures like grids or quadtrees, as the mutually applied forces effectively only affect vertices within a certain radius.

*e-mail: zellmann@uni-koeln.de
†e-mail: martin.weier@h-brs.de
‡e-mail: iwald@nvidia.com

In this paper, we show how the task of finding all vertices within a given radius can also be formulated as a ray tracing problem. This approach does not only create a simpler solution by leaving the problem of efficient data structure construction to the API, but also allows for leveraging hardware-accelerated NVIDIA RTX ray tracing cores (RT cores).

2 BACKGROUND AND PRIOR WORK

In the following, we provide background and discuss related work on force-directed graph drawing algorithms. We also give an introduction to NVIDIA RTX and prior work.

2.1 Force-directed graph drawing

We consider graphs $G = (V, E)$ with vertex set V and edge set E. Each $v \in V$ has a position $p(v) \in \mathbb{R}^2$. Edges $e \in E = \{u, v\}$, with $u, v \in V$, are undirected and unweighted. The Fruchterman-Reingold (FR) algorithm [9] (see Alg. 1) calculates the dispersion to displace each vertex based on the forces. A dampening factor is used to slow down the forces with an increasing number of iterations. Repulsive forces are computed for each pair of vertices $(u, v) \in V$. Attractive forces only affect those pairs that are connected by an edge. The following force functions are used:

$$F_{rep}(\Delta, k) = \frac{\Delta}{|\Delta|} \cdot \frac{k^2}{|\Delta|} \tag{1}$$

and

$$F_{att}(\Delta, k) = \frac{\Delta}{|\Delta|} \cdot \frac{\Delta^2}{|k|}, \tag{2}$$

where $\Delta = p(v) - p(u)$ is the vector between the two vertices acting forces upon each other. k is computed as $\sqrt{A/|V|}$, where A is the area of the axis-aligned bounding rectangle of V.

As the complexity of the first nested for loop per iteration is $O(|V|^2)$, and by observing that the pairwise forces diminish with increasing distance between vertices, the authors propose to adapt the computation of the repulsive force using:

$$F_{rep}(\Delta, k) = \frac{\Delta}{|\Delta|} \cdot \frac{k^2}{|\Delta|} u(2k - |\Delta|), \tag{3}$$

Algorithm 1 Fruchterman-Reingold spring embedder algorithm.

```
procedure SPRINGEMBEDDER(G(V,E),Iterations,k)
    for i := 1 to Iterations do
        D ← |V|                          ▷ dispersion to displace vertices
        for all v ∈ V do                 ▷ calculate repulsive forces (V x V)
            D(v) := 0
            for all u ∈ V do
                D(v) := D(v) + F_rep(p(v) − p(u), k)
            end for
        end for
        for all e ∈ E do                 ▷ calculate attractive forces
            D(v) := D(v) − F_att(p(v) − p(u), k)
            D(u) := D(u) + F_att(p(u) − p(v), k)
        end for
        for all v ∈ V do                 ▷ displace vertices according to forces
            DISPLACE(v,D(v),t)           ▷ t is a dampening factor
        end for
        t := COOL(t)                     ▷ Decrease dampening factor
    end for
end procedure
```

where $u(x)$ is 1 if $x > 0$ and 0 otherwise. With that, only vertices inside a radius $2k$ will have a non-zero contribution, which in turn allows for employing acceleration data structures to focus computations on only vertices within the neighborhood of $p(v)$.

The FR algorithm is a good match for GPUs as the three phases—repulsive force computation, attractive force computation, and vertex displacement—are highly parallel. The most apparent parallelization described by Klapka and Slaby [25] devotes one GPU kernel to each phase. The outer dimension of the nested for-loop over $v \in V$ is executed in parallel, but each GPU thread runs the full inner loop over $u \in V$ in Alg. 1. This reduces the time complexity to $\Theta(|V|)$, whereas the work complexity remains $\Theta(|V|^2)$. Force-directed algorithms—and in general graph drawing algorithms based on nearest neighbor search—lend themselves well to massive parallelization on distributed systems [1, 21] or on many-core systems and GPUs [17, 31, 33].

Gajdoš et al. [10] accelerate the repulsive force computation phase by initially sorting the $v \in V$ on a Morton curve. This order is subdivided into individual blocks to be processed in parallel in separate CUDA kernels. However, this process is inaccurate, as forces will only affect vertices from the same block. The authors try to account for that by randomly jittering vertex positions so that some of them spill over to neighboring blocks. Mi et al. [29] use a similar approximation but motivate that by imbalances originating from the multi-level approach described in [18] that they use in combination with FR. Our approach does not use approximations but is equivalent to the FR algorithm using the grid optimization that was proposed in the original work.

General nearest neighbor queries have been accelerated on the GPU with k-d trees, as in the work of Hu et al. [22] and by Wehr and Radkowski [37]. For dense graphs with $O(|E|) = O(|V|^2)$, the attractive force phase can also become a bottleneck. The works by Brandes and Pich [5] and by Gove [15] propose to choose only a subset of E using sampling to compute the attractive forces. Gove also suggests using sampling for the graph's vertex set V to improve the complexity of the repulsive force phase [16]. Other modifications to the stress model exist. The COAST algorithm by Ganser et al. [12] extends force-directed algorithms to support given, non-uniform edge lengths. They reformulate the stress function based on those edge lengths so that it can be solved using semi-definite programming. The maxent-stress model by Ganser et al. [13] initially solves the model only for the edge lengths and later resolves the remaining degrees of freedom via an entropy maximization model. The repulsive force computation in this work is based on the classical N-body model by Barnes and Hut [3] and uses a quadtree data structure for the all-pairs comparison. Hachul and Jünger [20] gave a survey of force-directed algorithms for large graphs. For a general overview of force-directed graph drawing algorithms, we refer the reader to the book chapter by Kobourov [26].

2.2 RTX ray tracing

NVIDIA RTX APIs allow the user to test for intersections of rays and arbitrary geometric primitives. This technique is often used to generate raster images. Here, bounding volume hierarchies (BVHs) help to reduce the complexity of this test, which is otherwise proportional to the number of rays times the number of primitives. The user supplies a *bounds program* so that RTX can generate axis-aligned bounding boxes (AABBs) for the user geometry and *build* a BVH. Now, a *ray generation program* can be executed on the GPU's programmable shader cores that will *trace* rays through the BVH using an API call. In the *intersection program*, which is called when rays hit the AABBs, the user can test for and potentially report an intersection with the geometry. A reported intersection will then be available in potential *closest-hit* or *any-hit* programs. RTX GPUs perform BVH traversal in hardware. When RTX calls an intersection program, hardware traversal is interrupted and a context switch occurs that switches execution to the shader cores.

RTX was recently used to accelerate visualization algorithms like direct volume rendering [30] or glyph rendering [39]. RT cores have, however, also been used for non-rendering applications, such as the point location method on tetrahedral elements presented by Wald et al. [36].

3 Method Overview

We propose to reformulate the FR algorithm as a ray tracing problem. That way, we can use an RTX BVH to accelerate the nearest neighbor query during the repulsive force computation phase. The queries and data structures used by the two algorithms differ substantially: force-directed algorithms use spatial subdivision data structures, whereas RTX uses object subdivision. Nearest neighbor queries do not directly map to the ray / primitive intersection query supported by RTX. However, we present a mapping from one approach to the other and demonstrate its effectiveness using an FR implementation with the CUDA GPU programming interface.

3.1 Mapping the force-directed graph drawing problem to a ray tracing problem

We present a high-level overview of our approach in Fig. 2. A nearest neighbor query can be performed by expanding a circle around the position $p(v)$ of the vertex $v \in V$ that we are interested in and *gathering* all $u \in V, u \neq v$ inside that circle. To compute forces, we would perform that search query for all $v \in V$ and would integrate the accumulation of the forces directly into the query.

By observing that the circle we expand around v always has a radius $2k$, we can *reverse* the problem: instead of expanding a circle around v, we instead expand circles around *all* $v \in V$. We then trace an *epsilon ray* with infinitesimal length and origin at $p(v)$ against this set of circles and accumulate the forces whenever $p(v)$ is inside the circle associated with $u \in V$, given that $u \neq v$. The intersection routine of the ray tracer only has to compute the length of the vector between the ray origin and the center of the circle and report an intersection whenever that length is less than $2k$. Geometrically, one can think of this as splatting, where the splats whose footprints overlap $p(v)$ act a repulsive force upon v.

The runtime complexity of the repulsive force computation phase using nearest neighbor queries can be reduced from $\Theta(|V|^2)$ to $\Theta(|V|\log(|V|))$ using spatial indices like quadtrees [18] or binary space partitioning trees [28] built over V. The spatial index would have to be rebuilt on each iteration. Likewise, the ray tracing query complexity can be reduced in the same manner using a BVH.

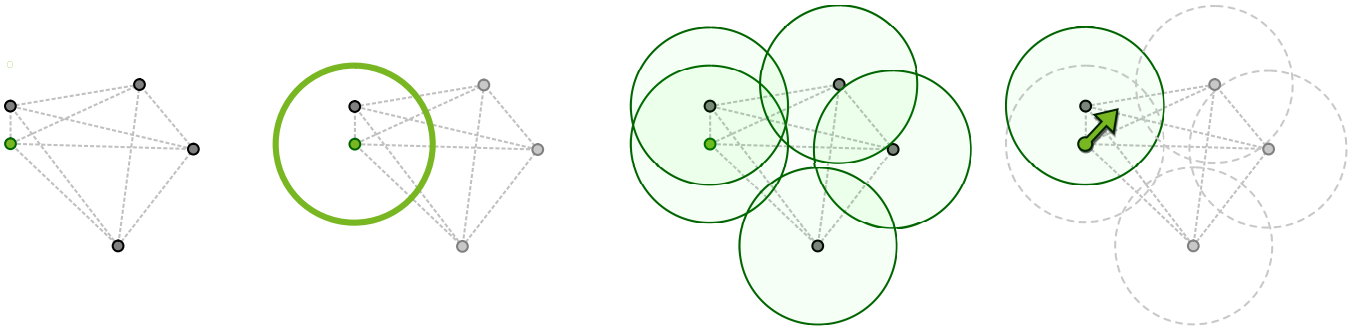

Figure 2: Mapping nearest neighbor queries to ray tracing queries. (a) The K_5: 10 graph; we are interested in the repulsive forces acted upon the green vertex by all the other vertices. (b) Nearest neighbor queries are performed by gathering the vertices inside a circle around the green vertex. (c) With a ray tracing query, instead of expanding a circle around the vertex of interest, we expand circles *around all vertices*. (d) We trace an *epsilon ray* (green arrow) originating at the green vertex' position and with infinitesimal length against the circles' geometry. Every circle that overlaps the ray origin—except the circle belonging to the vertex of interest itself—contributes to the force on the green vertex.

3.2 Implementation with CUDA and OptiX 7

We implemented the FR algorithm with CUDA. We use separate CUDA kernels for the repulsive and attractive forces and for the vertex dispersion phase. Those kernels are called sequentially in a loop over all iterations. The dispersion that is computed during the force phases is stored and updated in a global GPU array.

The parallel attractive force phase uses atomic operations to update the dispersion array. The repulsive phase is implemented using OptiX 7 and the OptiX Wrapper Library (OWL) [35]. Since the number of vertices will never change, we use a global, fixed-size GPU array for the 2-d positions that is shared between CUDA kernels and OptiX programs. Initial vertex placement is at random and in a square. RTX does not support 2-d primitives, so that we construct the BVH from discs with infinitesimal thickness.

The ray generation program spawns one infinitesimal ray per vertex v originating at $p(v)$; we again account for RTX being a 3-d API by setting the z coordinates of the ray origin and direction vector to 0 and 1, respectively. In this way, we can directly accumulate the dispersion inside the intersection program and do not even have to *report* an intersection that would otherwise be passed along to a potential closest-hit or any-hit program.

4 Evaluation

For a comparison with a fairly optimized, GPU-based nearest neighbor query, we use a 2-d spatial data structure based on the LBVH algorithm [27, 40]. As the vertices have no area, we obtain a 2-d BSP tree with axis-aligned split planes that subdivide parent nodes into two same-sized halves (*middle split*). With the restriction being relaxed that two split planes need to be placed at once, we should outperform the commonly used grid or quadtree implementations [6, 16]. Using Karras' construction algorithm [24], the build complexity is $O(n)$ in the number of primitives. Our motivation to use a data structure with superior construction performance is that is must be rebuild after each iteration. We use a full traversal stack in local GPU memory and perform nearest neighbor queries by gathering all vertices within a $2k$ radius around the current vertex position at the leaves. We have a slight advantage over RTX as our data structure is tailored for 2-d. At the same time we note that we cannot possibly optimize our data structure in the same way that NVIDIA probably has done with RTX, and neither that this is our goal with this comparison.

Note that the LBVH and RTX implementations and grid-based FR result in identical graph layouts. In comparison to state-of-the-art implementations in graph drawing libraries such as OGDF [6], Tulip [2], or Gephi [4]—all of which provide sequential CPU implementations of FR—both our RTX and LBVH solutions are magnitudes faster. In order to put both our GPU results into perspective, we also implemented the naive GPU parallelization from [25] over just the outer loop of the repulsive force phase.

We report execution times for the four data sets depicted in Table 1. Two artificial data sets consist of many fully connected K_5: 10 graphs (five vertices, ten edges). In one case we use $5K$ of those and sequentially connect pairs of them with a single edge. In the second case we use $50K$ of them as individual connected components. We also test using a complete binary tree with depth 16, as well as the graph representing twitter feed data that is also depicted in Fig. 1. For the results reported in Table 1 we used an NVIDIA GTX 1080 Ti (no RT cores), an RTX 2070, and a Quadro RTX 8000. The scalability study from Fig. 3 and the evaluation of the repulsive phase in Table 2 were conducted solely on the Quadro GPU.

5 Discussion

Our evaluation suggests speedups of $4\times$ to $13\times$ over LBVH. From the difference between the mean iteration times in Table 1 and the mean times for only the repulsive phase in Table 2 we see that the algorithm is dominated by the latter. The other phases plus overhead account for less than 1 % of the execution time. While Fig. 3 shows that our method's performance overhead for small graphs can be neglected—because it is on the order of about 1 ms——we observe dramatic speedups that increase asymptotically with $|V|$.

Interestingly, we see about the same *relative* speedups on the GeForce GTX GPU and on the RTX 2070 GPU with hardware acceleration. At the same time, we observe that the *absolute* runtimes differ substantially, which we cannot intuitively explain, as neither the peak performance in FLOPS, nor the memory performance of the two GPUs, differ that much. Profiling our handwritten CUDA nearest neighbor query, we find tree traversal to be limited by L2 cache hit rate, which is about 20 %. For RTX, such an analysis is impossible and we can only speculate about the results. It is conceivable that the RTX BVH has an optimized memory layout such as the one by Ylitie et al. [38]. Assuming that we are bound by memory access latency, the speedups we observe might stem from better utilization of the GPU's memory subsystem rather than hardware acceleration. Switching between hardware and software execution on RTX GPUs incurs an expensive context switch. Hardware traversal is interrupted whenever the intersection program is called. For our test data sets, we consistently found the average number of intersection program instances called to be in the hundreds. We might see an adversarial effect where we, on the one hand, benefit from hardware acceleration, but on the other hand suffer from expensive context switches and that the two effects in the end cancel. We find

Table 1: Statistics and average execution times on different GPUs. We use three artificial graphs with different connectivity and edge degrees, and a twitter feed graph. $c \in C$ denote connected components. Execution times reported are per full iteration including all phases.

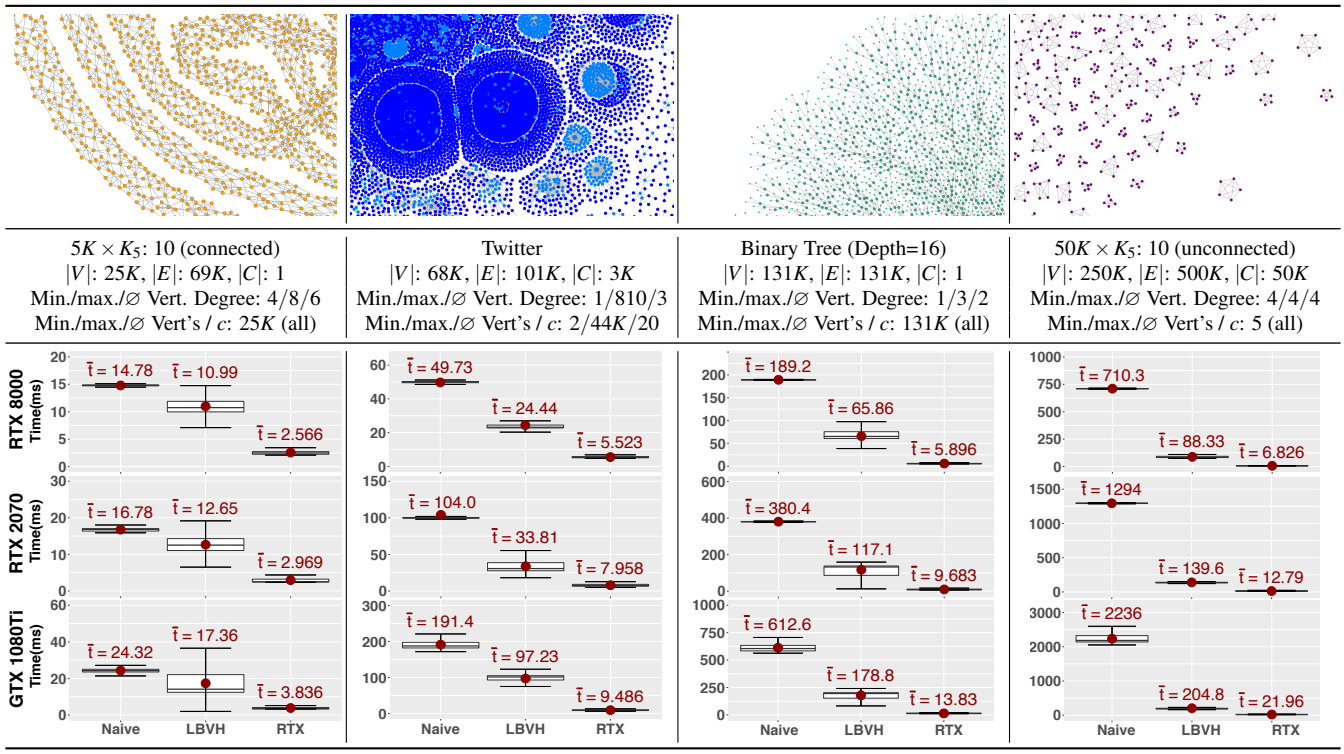

Table 2: Acceleration data structure statistics on RTX 8000, for the repulsive force computation phases. Execution times per iteration are given in milliseconds and the ratio of build vs. traversal times in percent. We also report total BVH memory consumption in MB.

Data Set	Mode	Mem	Build	Traversal	ΣF_{rep}	Speedup
$5K \times K_5$: 10 (connected)	LBVH	1.53	0.92 (8.37%)	10.0 (91.6%)	10.9	
	RTX	1.18	1.16 (45.5%)	1.39 (54.5%)	2.55	**4.27**×
Twitter	LBVH	4.16	1.94 (7.94%)	22.5 (92.1%)	24.4	
	RTX	3.22	2.18 (39.7%)	3.31 (60.3%)	5.49	**4.44**×
Binary Tree (Depth=16)	LBVH	8.00	2.53 (3.84%)	63.3 (96.2%)	65.8	
	RTX	6.19	2.36 (40.3%)	3.50 (59.7%)	5.87	**11.2**×
$50K \times K_5$: 10 (unconnected)	LBVH	15.3	2.87 (3.26%)	85.4 (96.7%)	88.3	
	RTX	11.8	2.82 (41.6%)	3.95 (58.4%)	6.77	**13.0**×

the speedups that we observe reassuring, especially because using RTX lifts the burden of having to program an optimized tree traversal algorithm for the GPU from the user.

6 Limitations of Our Study

We acknowledge that force-directed methods for large graphs exist that require fewer *iterations* to arrive at a converged layout and outperform FR by far in this regard [20] and are often based on multilevel optimizations [34]. We chose FR as a most *simple* force-directed algorithm to reason about the speedup and practicability of our approach. Algorithms that perform a nearest neighbor search to compute forces will generally benefit from the proposed techniques. The Fast Multipole Multilevel Method (FM^3) [19] employs such a nearest neighbor search and uses a coarsening phase in-between iterations. Similar to our method, the GPU multipole algorithm by Godiyal et al. [14] employs a k-d tree that is rebuilt per iteration, uses stackless traversal, and would likely benefit from RTX.

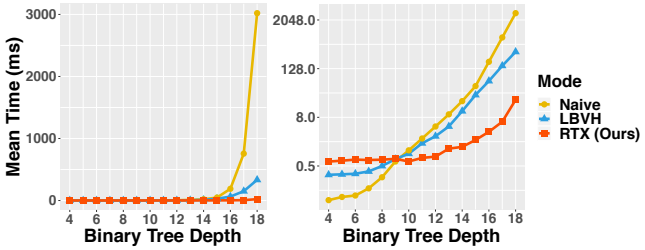

Figure 3: Scalability study where we build complete binary trees with depth $D = 4, 5, \ldots, 18$. Left: linear scale, right: logarithmic scale. We report mean times for only the repulsive force phase.

The GRIP method by Gajer and Kobourov [11] employs a refinement phase that *uses* FR to compute local displacement vectors. Although we assume that our approach will *complement* state-of-the-art algorithms with better convergence rates, a thorough comparison is outside of this paper's scope and presents a compelling direction for future work.

7 Conclusions

We presented a GPU-based optimization to the force-directed Fruchterman-Reingold graph drawing algorithm by mapping the nearest neighbor query performed during the repulsive force computation phase to a ray tracing problem that can be solved with RT core hardware. The speedup over a nearest neighbor query with a state-of-the-art data structure that we observe is encouraging. Force-directed algorithms lend themselves to a parallelization with GPUs. We found that those algorithms can be optimized even further by using RT cores and hope that our work raises awareness for this hardware feature even outside the typical graphics and rendering communities.

REFERENCES

[1] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani. A distributed multilevel force-directed algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):754–765, Apr. 2019. doi: 10.1109/tpds.2018.2869805

[2] D. Auber. Tulip - a huge graph visualization framework. In M. Jünger and P. Mutzel, eds., *Graph Drawing Software*, pp. 105–126. Springer, 2004.

[3] J. E. Barnes and P. Hut. A hierarchical O(n-log-n) force calculation algorithm. *Nature*, 324:446, 1986.

[4] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

[5] U. Brandes and C. Pich. Eigensolver methods for progressive multidimensional scaling of large data. In M. Kaufmann and D. Wagner, eds., *Graph Drawing*, pp. 42–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[6] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The Open Graph Drawing Framework (OGDF). In R. Tamassia, ed., *Handbook of Graph Drawing and Visualization*, chap. 15, pp. 543–569. CRC Press, Oxford, 2014.

[7] G. Di Battista. Graph drawing: the aesthetics-complexity trade-off. In K. Inderfurth, G. Schwödiauer, W. Domschke, F. Juhnke, P. Kleinschmidt, and G. Wäscher, eds., *Operations Research Proceedings 1999*, pp. 92–94. Springer Berlin Heidelberg, 2000.

[8] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[9] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. doi: 10.1002/spe.4380211102

[10] P. Gajdoš, T. Ježowicz, V. Uher, and P. Dohnálek. A parallel Fruchterman-Reingold algorithm optimized for fast visualization of large graphs and swarms of data. *Swarm and Evolutionary Computation*, 26:56 – 63, 2016. doi: 10.1016/j.swevo.2015.07.006

[11] P. Gajer and S. G. Kobourov. Grip: Graph drawing with intelligent placement. In J. Marks, ed., *Graph Drawing*, pp. 222–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[12] E. R. Gansner, Y. Hu, and S. Krishnan. COAST: A convex optimization approach to stress-based embedding. In S. Wismath and A. Wolff, eds., *Graph Drawing*, pp. 268–279. Springer International Publishing, 2013.

[13] E. R. Gansner, Y. Hu, and S. North. A maxent-stress model for graph layout. *IEEE Transactions on Visualization and Computer Graphics*, 19(6):927–940, 2013.

[14] A. Godiyal, J. Hoberock, M. Garland, and J. C. Hart. Rapid multipole graph drawing on the gpu. In I. G. Tollis and M. Patrignani, eds., *Graph Drawing*, pp. 90–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[15] R. Gove. Force-directed graph layouts by edge sampling. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 1–5, 2019.

[16] R. Gove. A random sampling O(n) force-calculation algorithm for graph layouts. *Computer Graphics Forum*, 38(3):739–751, 2019. doi: 10.1111/cgf.13724

[17] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227(18):8290 – 8313, 2008. doi: 10.1016/j.jcp.2008.05.023

[18] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In J. Pach, ed., *Graph Drawing*, pp. 285–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[19] S. Hachul and M. Jünger. Large-graph layout with the fast multipole multilevel method. Technical report, Zentrum für Angewandte Informatik Köln, 2005.

[20] S. Hachul and M. Jünger. Large-graph layout algorithms at work: An experimental study. *Journal of Graph Algorithms and Applications*, 11(2):345–369, 2007.

[21] A. Hinge, G. Richer, and D. Auber. Mugdad: Multilevel graph drawing algorithm in a distributed architecture. In *Conference on Computer Graphics, Visualization and Computer Vision*, p. 189. IADIS, Lisbon, Portugal, 2017.

[22] L. Hu, S. Nooshabadi, and M. Ahmadi. Massively parallel kd-tree construction and nearest neighbor search algorithms. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2752–2755, 2015.

[23] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7 – 15, 1989. doi: 10.1016/0020-0190(89)90102-6

[24] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pp. 33–37. Eurographics Association, Goslar Germany, Germany, 2012. doi: 10.2312/EGGH/HPG12/033-037

[25] O. Klapka and A. Slaby. nVidia CUDA platform in graph visualization. In S. Kunifuji, G. A. Papadopoulos, A. M. Skulimowski, and J. KacprzykÂ Â , eds., *Knowledge, Information and Creativity Support Systems*, pp. 511–520. Springer International Publishing, 2016.

[26] S. G. Kobourov. Force-directed drawing algorithms. In R. Tamassia, ed., *Handbook of Graph Drawing and Visualization*, chap. 12, pp. 383–408. CRC Press, Oxford, 2014.

[27] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 2009. doi: 10.1111/j.1467-8659.2009.01377.x

[28] U. Lauther. Multipole-based force approximation revisited – a simple but fast implementation using a dynamized enclosing-circle-enhanced k-d-tree. In M. Kaufmann and D. Wagner, eds., *Graph Drawing*, pp. 20–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[29] P. Mi, M. Sun, M. Masiane, Y. Cao, and C. North. Interactive graph layout of a million nodes. *Informatics*, 3(4):23, 2016.

[30] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *2019 IEEE Visualization Conference (VIS)*, pp. 256–260, Oct 2019. doi: 10.1109/VISUAL.2019.8933539

[31] A. Panagiotidis, G. Reina, M. Burch, T. Pfannkuch, and T. Ertl. Consistently gpu-accelerated graph visualization. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*, VINCI '15, p. 35–41. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2801040.2801053

[32] H. C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501 – 516, 2002. doi: 10.1006/jvlc.2002.0232

[33] V. Uher, P. Gajdo, and V. Snáel. The visualization of large graphs accelerated by the parallel nearest neighbors algorithm. In *2016 IEEE Second International Conference on Multimedia Big Data (BigMM)*, pp. 9–16, 2016.

[34] A. Valejo, V. Ferreira, R. Fabbri, M. C. F. d. Oliveira, and A. d. A. Lopes. A critical survey of the multilevel method in complex networks. *ACM Comput. Surv.*, 53(2), Apr. 2020. doi: 10.1145/3379347

[35] I. Wald, N. Morrical, and E. Haines. OWL – The Optix 7 Wrapper Library, 2020.

[36] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In M. Steinberger and T. Foley, eds., *High-Performance Graphics - Short Papers*. The Eurographics Association, 2019. doi: 10.2312/hpg.20191189

[37] D. Wehr and R. Radkowski. Parallel kd-tree construction on the GPU with an adaptive split and sort strategy. *Int. J. Parallel Program.*, 46(6):1139–1156, Dec. 2018. doi: 10.1007/s10766-018-0571-0

[38] H. Ylitie, T. Karras, and S. Laine. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In V. Havran and K. Vaiyanathan, eds., *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2017. doi: 10.1145/3105762.3105773

[39] S. Zellmann, M. Aumüller, N. Marshak, and I. Wald. High-Quality Rendering of Glyphs Using Hardware-Accelerated Ray Tracing. In S. Frey, J. Huang, and F. Sadlo, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2020. doi: 10.2312/pgv.20201076

[40] S. Zellmann, M. Hellmann, and U. Lang. A linear time BVH construction algorithm for sparse volumes. In *Proceedings of the 12th IEEE Pacific Visualization Symposium*. IEEE, 2019.