Exploiting GPUs for fast force-directed visualization of large-scale networks

Govert G. Brinkmann*, Kristian F. D. Rietveld * and Frank W. Takes*†
* *LIACS, Leiden University, The Netherlands, E-mail:* {krietvel,ftakes}@liacs.nl
† *CORPNET, University of Amsterdam, The Netherlands*

Abstract*—Network analysis software relies on graph layout algorithms to enable users to visually explore network data. Nowadays, networks easily consist of millions of nodes and edges, resulting in hours of computation time to obtain a readable graph layout on a typical workstation. Although these machines usually do not have a very large number of CPU cores, they can easily be equipped with Graphics Processing Units (GPUs), opening up the possibility of exploiting hundreds or even thousands of cores to counter the aforementioned computational challenges. In this paper we introduce a novel GPU framework for visualizing large real-world network data. The main focus is on a GPU implementation of force-directed graph layout algorithms, which are known to create high quality network visualizations. The proposed framework is used to parallelize the well-known ForceAtlas2 algorithm, which is widely used in many popular network analysis packages and toolkits. The different procedures and data structures of the algorithm are adjusted to the CUDA GPU architecture's specifics in terms of memory coalescing, shared memory usage and thread workload balance. To evaluate its performance, the GPU implementation is tested using a diverse set of 38 different large-scale real-world networks. This allows for a thorough characterization of the parallelizable components of both force-directed layout algorithms in general as well as the proposed GPU framework as a whole. Experiments demonstrate how the approach can efficiently process very large real-world networks, showing overall speedup factors between 40× and 123× compared to existing CPU implementations. In practice, this means that a network with 4 million nodes and 120 million edges can be visualized in 14 minutes rather than 9 hours.

***Keywords*-network visualization; force-directed graph layout; large-scale networks; parallel programming; CUDA;**

I. Introduction

Visualizing data allows the user to manually explore the represented information, and can aid tremendously in finding for example patterns and outliers. Here, the focus is on visualizing *networks* (or graphs) consisting of nodes, representing entities, and edges (or links), representing the relationships between these entities. To visualize a network, and thus the underlying data, one creates a drawing in the plane, depicting nodes as circles, and edges as lines between connected nodes. The main challenge in realizing such a drawing, is positioning the nodes in such a way that a 'useful' or 'readable' layout emerges that allows the viewer to perceive the structure of the graph. Several layout principles are generally considered to contribute to such layouts [1], such as minimizing edge crossings, preventing long edges and limiting overlap between nodes. Over the past decades, numerous graph layout algorithms (further discussed in Section III) have been introduced. Given the topology of a graph, these algorithms aim at computing a layout that adheres to common graph layout readability criteria [2]. Given that real networks are mostly non-random, not too densely connected and, to a certain extent, partitionable, these algorithms can generally produce a useful, readable and interpretable layout.

However, commonly used data originating from for example social networks, webgraphs, information networks and communication networks consists of millions of nodes and edges, resulting in major visualization challenges [3] in terms of readability. This readability issue is not the topic of this paper as it has largely been addressed by modern force-directed algorithms such as ForceAtlas2 [4]. This layout algorithm is widely used in network analysis toolkits such as Gephi [5] and is generally able to overcome local minima in the quality of the visualization, resulting in meaningful and readable visualizations. See Figure 2 for an example. More importantly, when larger networks are considered, algorithms must scale well in terms of time and memory usage. Although memory usage is typically linear in the number of nodes and edges (and thus acceptable), time consumption is a challenge. Using force-directed algorithms to visualize networks with more than a few hundred thousand nodes easily takes several hours of computation time, and is currently not feasible on workstations using available software packages. Solving these computational challenges is highly relevant for the network analysis community and hence the topic of this paper.

Given that workstation computers typically have four to eight processing cores at most, we will assess whether another component of workstations could be used to significantly speed up the graph layout process: the Graphics Processing Unit (GPU). The emergence of General Purpose computing on Graphics Processing Units (GPGPU) allowed many data-parallel algorithms to scale to significantly larger input [6], [7]. This results from the (massively) parallel architecture of GPUs, which is designed to concurrently transform billions of pixels per second. It is only recently that the applicability of GPGPU to graph algorithms has been studied, given that the irregular memory access patterns associated with such algorithms were initially considered challenging for the architecture of most GPUs [8]. However, the (embarrassingly) parallel character of most force-directed graph layout algorithms, in which node-based calculations are performed independently of each other, suggest they are well suited to be run on the parallel platform provided by GPUs (see e.g. [9] and the discussion of previous and related work in Section III).

In this paper, we present a GPU framework which implements the different components of force-directed layout algorithms on a GPU. We evaluate if and how the parallel architecture of GPUs can be used to reduce the computation time of layouts of graphs with millions of nodes and edges. Using a large number of datasets of real-world networks we assess the performance of our GPU implementation to determine its feasibility for large-scale network visualization. In contrast to earlier studies on the scalability of graph layout algorithms using GPUs [9], [10], [11], our focus is entirely on force-directed algorithms for real-world (social) networks. These networks are typically non-random, sparse, exhibit a power law degree distribution, have dense clusters, low average pairwise distances, and above all, are large in terms of the number of nodes and edges.

The remainder of this paper first introduces the concepts, notation and context of the problem in Section II. Next, related work on graph layout is discussed in Section III. The newly proposed GPU framework, including our implementation of ForceAtlas2 on the GPU, is the topic of Section IV, after which we present our data, experiments and results in Section V. Section VI concludes the paper and gives suggestions for future work.

II. Preliminaries

This section briefly reviews the graph theoretic notation and concepts we use throughout this paper, as well as a short introduction to GPU programming using CUDA.

A. Networks

A *graph* (or *network*) $G = (V, E)$ consists of a set of *vertices* (or *nodes*), V, and a set of *edges* (or *links*), E. In this paper we only consider undirected networks, as in visualizing graphs, link direction is usually ignored and simply incorporated by replacing lines by directed arrows.

Node u is adjacent to node v iff $\{u, v\} \in E$. Adjacent nodes are also called neighbors. A path from node u to node v is a sequence of nodes such that each of the subsequent nodes in the sequence are adjacent. The length of a shortest path is called the *distance* between node u and node v, denoted $d(u, v)$. The *degree* $deg(u)$ of a node $u \in V$ equals the number of nodes adjacent to u, i.e., $|\{v \mid \{u, v\} \in E\}|$.

A graph is connected if there exists a path between every pair of nodes. Given a graph $G = (V, E)$, the subgraph induced by $V' \subseteq V$, is the graph $G' = (V', E')$ obtained by taking $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. A connected component of G is a connected subgraph of G of maximal size, i.e., it cannot be extended by adding another node. Here we focus on the largest connected component, called the *giant component*, as a layout for the entire graph can simply be computed by merging the layouts obtained for each component.

In this paper we concern ourselves with *real-world networks* [12]. This means that the networks are sparse; they have few edges compared to the maximum number of edges. There is typically one giant component comprising the majority of the nodes. Degrees follow a power-law distribution with many peripheral low degree nodes and a small number of high degree hubs. The node clustering coefficient, which indicates the fraction of closed triangles among a node's adjacent nodes is, averaged over all nodes, very high, indicating the presence of tightly knitted groups of nodes in the network. Altogether, this results in very low average node-to-node distances, referred to as the small world property [12].

B. GPU Architectures and Programming Platforms

Over the past decade, GPUs gained prominence as coprocessors to CPUs, to aid in solving many data-parallel problems [6]. The arrival of GPGPU programming frameworks, such as OpenCL [13] and CUDA [14], that do not require programmers to reformulate their problem in terms of a computer graphics problem, accelerated this development. GPUs are ideally suited to tackle parallel problems, as GPUs dedicate more silicon to functional units that perform data manipulations. They optimize for throughput, in contrast to traditional CPU architectures which optimize for latency at the expense of elaborate control logic for features such as branch-prediction and out-of-order instruction execution.

The framework described in this paper has been implemented for NVIDIA GPUs using the CUDA platform [14]. GPU programming rests on the specification of compute *kernels*, which are the subroutines that are executed in parallel on the GPU using many threads of execution. To execute a kernel, a *block* of threads is formed, of which the threads are distributed across the GPU cores. In order to optimize the execution of such kernels on the GPU, characteristics of the GPU architecture have to be taken into account. Section IV-B will discuss how we have tailored our GPU implementation towards these characteristics.

On CUDA architectures, cores are grouped in *Streaming Multiprocessors (MPs)*. The threads of a block are, in turn, subdivided into multiple *warps* of 32 threads each. The scheduling unit of the multiprocessor is a warp, meaning that the threads of a warp always co-reside on a single MP. The notion of a warp is important, as threads in a warp execute in lock-step through a Single Instruction Multiple Threads (SIMT) architecture that advances them at the same time unless branching occurs, causing threads to execute serially until the next common instruction. Such branch-divergence is detrimental to performance and should be prevented. Furthermore, memory accesses made by a warp of threads are most efficient if they can occur in a coalesced manner. The GPU used in our experiments (for details, see Section

V), coalesces the memory accesses by a warp to consecutive words of 4-bytes, starting at an address that is a multiple of 32 bytes. The consecutive data is then fetched from memory using a strongly reduced number of transactions. For more details on performance optimization for CUDA GPUs, we refer the reader to [15].

III. Related Work

One of the first motivations behind the development of graph layout algorithms, was the the need for automatically generated flowcharts of algorithms and software [16]. In that context, Tutte [17] proposed a layout algorithm for 3-connected planar graphs which was one of the first 'force-directed' graph layout algorithms [18]. In this paper, we focus on modern *force-directed* layout algorithms. Several other approaches to the graph layout exist and we refer the reader to [1], [3], [19] for a comprehensive overview.

The force-directed as well as the associated spring-electrical model were first pioneered in [20], [21], [22]. It approaches the layout problem as an n-body problem by considering the layout as a physical system in which nodes, analogous to bodies, impose forces on each other, causing them to displace. Force-directed algorithms mostly vary in the force-model they use. While forces were initially chosen to replicate real-world physical systems, such as springs [20], this did not seem necessary to obtain good layouts [21]. Since then, artificial force-models have found their use in various algorithms [4]. Some algorithms introduce additional constraints, next to the force model previously described to improve the quality of the resulting visualization [23]. Early force-directed algorithms suffered from sub-optimal layouts given large graphs as input [3]. This was caused by the fact that increasing the size of the graph introduces many local minima in which the layout algorithm can get stuck. Therefore, usually a 'speed' or 'temperature' scales the displacement of nodes during the layout process, as discussed in Section IV-B. Since force-directed algorithms involve computing repulsive forces between all node pairs, time complexity is $\mathcal{O}(n^2)$. Fruchterman et. al [21] made an effort to overcome this quadratic complexity by proposing a 'grid-based' algorithm in which the layout space is partitioned using a grid and repulsive forces are only calculated between nodes in neighboring grid cells. Other approaches include the Fast Multipole Method (FMM) [24], [25] and the use of Barnes-Huts approximation [26] as further discussed in Section IV-B.

Other types of algorithms including multilevel approaches have been implemented on the GPU [9], [10], [11]. However in this paper the main focus is on the use of GPUs for *force-directed* visualization of non-artificial, real-world data. Our main goal is to create a fast parallel implementation of the high quality force-directed visualization algorithm ForceAtlas2 [4], exploiting the parallel aspect of GPUs, such that networks with millions of nodes can be visualized.

IV. Proposed GPU Framework

This section proposes our framework for the implementation of force-directed network visualization algorithms on the GPU. Within this framework the different steps of graph visualization algorithms are broken down into separate components. Together, a sequence of components forms a pipeline. This allows new visualization algorithms to be swiftly implemented by creating a new pipeline, re-using existing components and only writing a minimal amount of new code. Additionally, it is straightforward to interchange individual components with other components that achieve the same goal, for example using a faster algorithm or an algorithm optimized for particular graph properties. In this paper, the focus is on the implementation of the framework's components used by force-directed approaches to graph layout. In Section IV-A we introduce the different components that are distinguished within the framework. Section IV-B discusses how the popular ForceAtlas2 algorithm [4] is implemented on the GPU using this framework, specifically taking into account architectural characteristics of GPUs.

A. Components

The components within the framework have been developed to accommodate the swift implementation of force-directed graph layout algorithms. As discussed, force-directed approaches to graph layout consider the problem of graph layout as an n-body problem, in which nodes impose forces on each other causing them to displace. Initially, all nodes are randomly positioned in a rectangular area of the Euclidean plane. Subsequently, the graph layout process is performed using the following five framework components:

1) *Gravity*: a component that applies a gravitational force, towards the origin of the layout space, on each node. This force ensures that disconnected components or remote node groups do not 'drift' to the periphery of the drawing.
2) *AttractiveForce*: this component computes attractive forces that are induced between neighboring nodes. This enforces that related nodes are positioned in proximity of each other, whilst unrelated nodes are positioned at a distance.
3) *BodyRepulsion*: computes the repulsive forces exerted between every pair of nodes. As outlined in the preceding section, different algorithms can be employed to do so, which can be implemented as different variants of this component.
4) *UpdateSpeed*: after force computation, there is the possibility to update parameters that depend on these results. This component can be optionally introduced in the pipeline to compute and update global speed.
5) *Displacement*: displaces the nodes based on the net forces exerted on them. The displacement is typically scaled using a 'speed' or 'temperature' parameter to

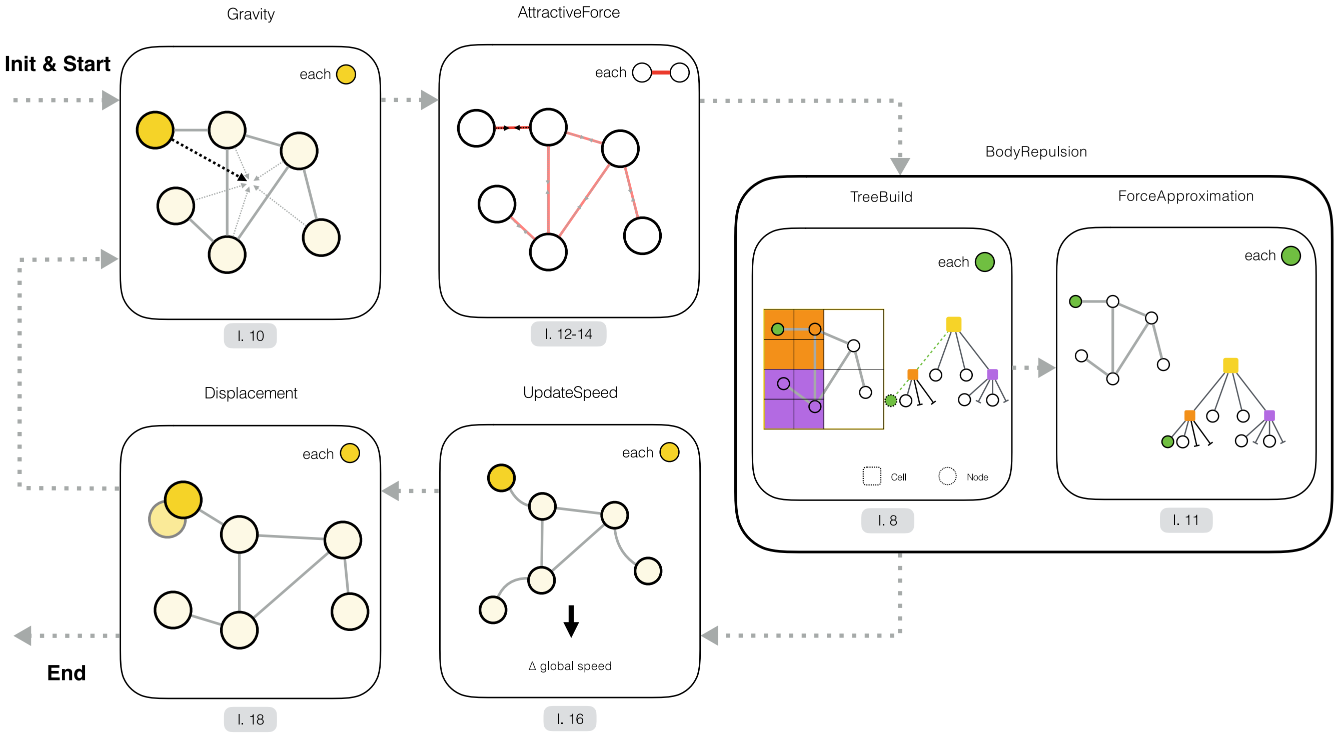

Figure 1. Illustration of the different components, together forming a graph visualization pipeline that is executed on the GPU. The top-right of each illustration indicates whether each thread (concurrently) executing the component operates on nodes or edges. After initialization, execution starts at the Gravity component. The gray boxes on the bottom indicate the correspondence of the components with the lines in Algorithm 1.

enable convergence to a (locally) optimal configuration and to account for certain graph properties.

All steps are repeated until a given maximum number of iterations is reached, or a stopping criterion is met, at which time the algorithm is terminated. The different steps are summarized in Figure 1. The main strength of this framework approach is the ability to re-use and interchange components, which allows rapid and dynamic implementation of graph visualization algorithms. While some components perform all computations per node, other components iterate over edges or node pairs (see top right corner of components in Figure 1). Without affecting the final result, a component may be interchanged with another component performing the same computation but using a different iteration pattern, beneficial and optimized for a certain class of graphs. Furthermore, apart from tuning the iteration pattern, different variants of the *BodyRepulsion* component may be provided that implement different algorithms.

The framework also allows for flexibility in implementing graph layout algorithms. For instance, not all algorithms apply a gravitational force, so this step can sometimes be skipped. The *Displacement* kernel can be made to work with single or multiple speeds. In general, a decreasing speed causes the layout to converge to a (locally) optimal configuration. Commonly, global speeds, affecting the entire layout, as well as local speeds, affecting groups of nodes or individual nodes, are used. Local speeds are beneficial because they can be used to prevent individual nodes, or groups of nodes, from oscillating around the same position [4]. Furthermore, new components can be introduced to the pipeline, for example to introduce forces exerted on nodes or edges that are dependent on particular graph properties. This flexibility persists as long as these routines allow the force for all nodes or edges to be computed in parallel.

Our framework reads the input graphs using the common and widely used edge list file format. This allows for effortless integration of the framework into existing network analysis software. The output of the framework consists of a node list with coordinate pairs, which can be re-used in visualization tools to draw the graph on screen or to write a vector graphics file (such as SVG) to disk. For OpenGL applications, there exists the possibility to directly store the results of the CUDA computations in OpenGL buffers, without incurring the CPU-GPU data transfer penalty, allowing the graph visualization process to be displayed in an interactive real-time environment.

B. *Implementation of ForceAtlas2*

To illustrate how our framework can be used to implement graph visualization algorithms, we consider the ForceAtlas2 algorithm [4]. ForceAtlas2 is a force-directed layout algorithm developed for Gephi [5], an open-source tool for social network analysis. The main contribution of ForceAtlas2 to research on force-directed algorithms is its force model and its implementation of adaptive speed. We have chosen this particular algorithm since its operation is characteristic of force-directed layout algorithms and because the choice of force model makes ForceAtlas2 well suited to visualize real-world networks, which is our primary topic of interest.

The ForceAtlas2 algorithm is described in the pseudo-code in Algorithm 1. For more details, the reader is referred to [4]. Note that in each layout iteration the same operation, force computation, is performed on the nodes which are independent data elements. It is this kind of data-parallel problems for which the architecture of GPUs is specifically suited, typically leading to significant performance improvements for these kinds of problems [6]. The relationship of the sequential ForceAtlas2 algorithm and the components of our framework can be seen by comparing the line numbers in pseudo-code with the line numbers seen in the component labels of Figure 1. An example visualization created using ForceAtlas2 is given in Figure 2.

The components of the framework have been implemented using the CUDA C environment [27], an extension of the C programming language. Each component consists of one or more CUDA kernels (for details, see Section II-B). The different CUDA kernels are launched one after the other, in thread blocks consisting of many (> 1000) threads. Each thread will process a small subset of the data. This data can be either nodes or edges, depending on the kernel that is considered (see Figure 1). We launch at least one block per multiprocessor, but usually more. All data to be processed by each kernel is present in global memory of the GPU, such that no transfers between CPU memory and GPU memory, and associated delays, occur as the components are executed. All data is stored in global memory to allow for fully coalesced memory accesses: aligned to 32-byte boundaries and stored at consecutive memory addresses. Structure types, e.g. node positions in the plane with an x and a y component, are flattened from arrays of structures to structures of arrays storage. This way, structure members are not stored interleaved but consecutively, allowing for fully coalesced memory access when all threads access the same component of consecutive nodes. We now detail for each of the framework components mentioned in Section IV-A how these have been tailored for execution on the GPU.

1) Gravity Component: The gravity component applies a gravitational force to each node, which is proportional to its distance to the center of the layout, as detailed in [4]. Each thread processes a group of nodes and due to the aforementioned data structure of the positional data, all memory accesses are fully coalesced.

2) AttractiveForce Component: The attraction kernel applies an attractive force between all nodes connected by an edge. To avoid thread divergence, each thread processes edges, instead of nodes. As nodes in real-world networks typically have different degrees, a node-parallel implementation would imply different workloads for each thread. Still, in order to prevent race conditions, an edge-parallel implementation does imply we need to use atomic operations to update forces on the nodes comprising an edge. The edges have been stored as a structure of arrays, such that there is an array containing all sources and a separate array containing all targets. The order of the nodes in the sources array matches that of the position and force arrays, making all memory accesses for the source node coalesced, but all memory accesses for the target node not coalesced.

Algorithm 1 ForceAtlas2 [4]

```
    Input: Undirected graph $G = (V, E)$, $iterations$, gravitational and repulsive force scalars $f_g$ and $f_r$.
    Output: A position $\mathbf{p}_v \in \mathbb{R}^2$ for each $v \in V$.

 1: $global\_speed \leftarrow 1.0$
 2: for all $v \in V$ do                                   ▷ Initialize variables
 3:     $\mathbf{p}_v = random()$
 4:     $\mathbf{f}_v = (0.0, 0.0)^\top$                    ▷ Net force on node $v$
 5:     $\mathbf{f}'_v = (0.0, 0.0)^\top$      ▷ $\mathbf{f}'_v$ is $\mathbf{f}_v$ of preceding iteration
 6: end for
 7: for $i = 1 \rightarrow iterations$ do
 8:     BH.rebuild()                         ▷ (Re)build Barnes-Hut tree
 9:     for all $a \in V$ do
10:         $\mathbf{f}_v \leftarrow \mathbf{f}_v - \mathbf{p}_v$                 ▷ (Strong) Gravity
11:         $\mathbf{f}_v \leftarrow \mathbf{f}_v + k_r \cdot BH.force\_at(\mathbf{p}_v)$   ▷ Repulsion
12:         for all $w \in neighbors(v)$ do
13:             $\mathbf{f}_v \leftarrow \mathbf{f}_v + \frac{\mathbf{p}_v - \mathbf{p}_w}{|\mathbf{p}_v - \mathbf{p}_w|}$            ▷ Attraction
14:         end for
15:     end for
16:     $UpdateGlobalSpeed()$
17:     for all $v \in V$ do
18:         $\mathbf{p}_v \leftarrow local\_speed(v) * \mathbf{f}_v$       ▷ Displacement
19:         $\mathbf{f}'_v \leftarrow \mathbf{f}_v$
20:         $\mathbf{f}_v \leftarrow (0.0, 0.0)$
21:     end for
22: end for

23: function LOCAL_SPEED($v$)                          ▷ for a node $v$
24:     return $\frac{global\_speed}{1.0 + \sqrt{global\_speed + swing(v)}}$
25: end function

26: function SWING($v$)                                ▷ for a node $v$
27:     return $|\mathbf{f}_n - \mathbf{f}'_n|$
28: end function
```

3) BodyRepulsion Component: ForceAtlas2 uses Barnes-Hut approximation [26] to determine repulsive forces between all node-pairs in $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$ time.

In a nutshell, the Barnes-Hut algorithm divides the space into cubic cells. The root cell, comprising the entire space, is recursively divided into subcells (of equal size) until no cell holds more than one body. The tree that describes this recursive structure is then used to approximate the forces induced on the different bodies. For details, see [26].

We used the CUDA C implementation of the Barnes-Hut algorithm described (and provided) by Burtscher and Pingali [28], which consists of multiple CUDA kernels to implement this algorithm. The implementation has been slightly modified by simplifying the implementation from a three-dimensional to a two-dimensional one, given that we use a two-dimensional layout space for our graph layout. Note that in this case this component consists of two kernels: one to build the tree and one to approximate the forces.

4) UpdateSpeed Component: The majority of work in updating the global speed is summing swing and traction values over all nodes (see [4] for a more detailed description). To do so, each thread processes a single node, and computes its swing values and traction value. We use a reduction in the shared memory of the GPU multiprocessors, as exemplified in Section B.5 of the CUDA C guide [27], to combine these values into global swing and traction values.

5) Displacement Component: This component displaces all nodes, depending on the force induced on them. Each thread processes a single node, and all memory accesses are fully coalesced. No thread divergence occurs.

V. Experiments

In this section, we evaluate the performance of our GPU framework. First, the experimental setup is described in Section V-A. The diverse set of large-scale real-world networks described in Section V-B is then used to conduct experiments, of which the results are reported in Section V-C and discussed in Section V-D.

A. Experimental setup

For our experiments we used an NVIDIA GTX Titan X graphics card, containing the NVIDIA GM200 GPU clocked at 1 GHz. It is based on the Maxwell architecture, and contains 24 Streaming Multiprocessors of 128 cores each, summing to a total of 3,072 available CUDA cores. The CPU implementation was executed on an Intel Xeon E5-2650 v3 CPU clocked at 2.3 GHz. All datasets fit in main memory of the GPU and CPU, thus RAM or disk usage is not relevant. All code was compiled using the `-O3` optimization flag, using GCC (v. 4.8.5) and NVCC (v. 7.5.17).

The main goal of the experiments is to assess whether the computation time of visualizing large-scale networks can be reduced using the GPU. Given that the Java implementation of the original ForceAtlas2 algorithm in the Gephi toolkit was not capable of visualizing networks of this size, a C++ implementation was developed by a direct translation of the Java algorithm to C++. This CPU implementation always performs on par with (but often a lot better than) the implementation in the Gephi toolkit in terms of computation time. It was therefore also directly used as the basis for the CUDA GPU implementation. As to our knowledge no other high-performance implementations of the ForceAtlas2 algorithm exist, we used our sequential C++ implementation as the baseline for the performance of the GPU implementation. All code used to (re)produce the results presented in this paper can be found at https://liacs.leidenuniv.nl/~takesfw/GPUNetworkVis.

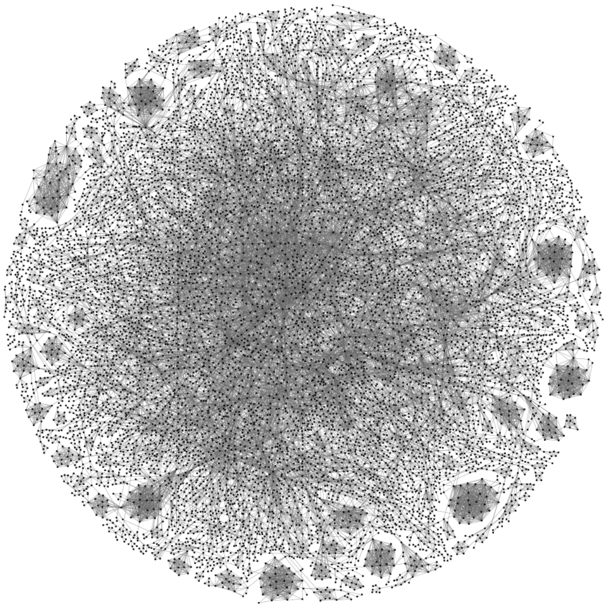

Figure 2. Visualization of a sample of the CORPNET$_3$ dataset (see Section V-B) with $10,000$ nodes and $27,000$ edges. Created using ForceAtlas2 (with stronger gravity, $f_g = 1$, $f_r = 80$).

Finally we recall that the quality of the visualization is not relevant in these experiments, as exactly the same algorithm is used in both the CPU and GPU implementation. Although in theory the concurrent updates on node coordinates in the GPU implementation could cause small layout divergences compared to the sequential implementation, this effect is insignificant considering the random initialization of the layout at the start of the algorithm.

For our implementation of ForceAtlas2, we have chosen to apply ‘strong gravity’ to nodes. This, in contrast to the regular gravity model, is not proportional to the distance between the considered node and the the origin. Of course, the regular gravity model can be used by replacing the gravity component. We used the default attractive force, which is proportional to the distance between nodes (in the layout). Gravitational force was not scaled ($f_g = 1$). Repulsive force was scaled by a factor 80 ($f_r = 80$) to compensate for the overlap between nodes that resulted from the use of 'strong gravity'. Note that this parameter is mainly to tune the aesthetics of the final visualization, and has no significant effect on the performance of the algorithm.

B. Data

To ensure that our implementation is generic and not biased towards certain types of networks, we used a number of real-world network datasets, gathered from KONECT [29], a large repository of real-world network datasets. The selection contains social networks, information networks, webgraphs, physical router networks, e-mail communication networks, movie actor co-occurrence networks, webgraphs and scientific collaboration and citation networks. In addition, three large-scale networks consisting of millions of nodes and edges were especially created for this study.

The GITHUB dataset was created from the online platform GitHub that allows programmers to collaborate on the development of software. Users can contribute to 'repositories' containing the code and other resources related to a software project. Using data from the GitHub Archive, we constructed a 'collaboration graph' in which distinct repositories are connected if they share contributors (users). The resulting network gives us insight in how projects on GitHub are related, based on whether they share developers.

Datasets CORPNET$_3$ and CORPNET$_4$ represent a corporate network. Corporations around the world interact with each other in many different ways, including trade, ownership and by means of interlocking the directorates. The latter so-called board interlock networks capture interaction between companies at the governance level: nodes are companies and edges represent shared board members or directors between companies. For more information on the analysis of these types of networks, see [30]. Two versions of this data are used: CORPNET$_3$ and CORPNET$_4$, consisting of 3.1 million and 4.5 million nodes, respectively.

The first seven columns of Table I list the names as well as a number of basic network properties (see Section II-A for definitions) of the giant components of each of the in total 38 considered networks datasets.

C. Results

In addition to the structural properties of the considered network datasets, Table I lists the speedup achieved by the GPU implementation compared to sequential baseline CPU execution for each of the six kernels, followed by the overall speedup in bold in the column entitled "Total". Recall from Section IV-B that although we have five components, the *BodyRepulsion* component is split over two kernels. The second to last column indicates the (wall-clock) execution time to execute 500 iterations of all of the CUDA kernels in minutes (which is typically more than enough for convergence to a readable layout, see [4]). The time necessary to transfer data to and from the GPU has not been included, as this is negligible compared to the computational time used. In the last column of Table I we also list the average execution time of the equivalent CPU implementation, to serve as a reference of the performance of existing implementations of force-directed algorithms.

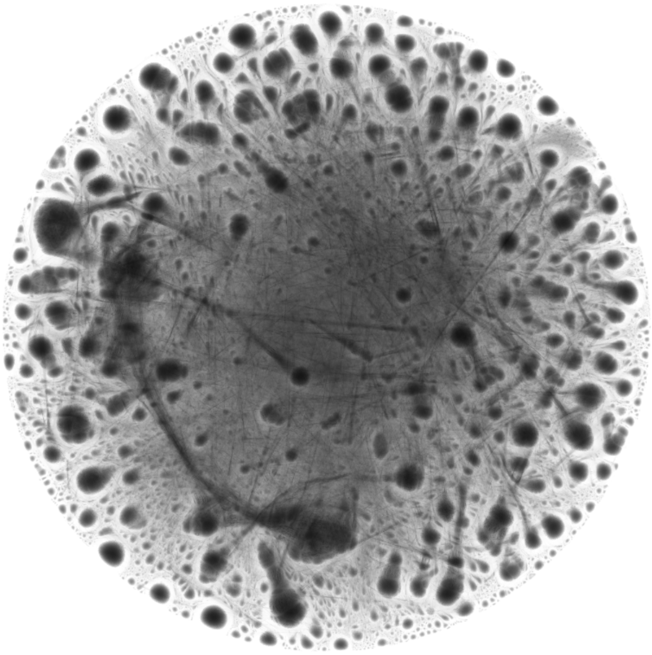

Figure 3. Resulting layout for the CORPNET$_4$ network with $4,602,225$ nodes and $123,329,543$ edges.

To evaluate the performance of the six individual components, execution times were determined over the first ten iterations of the algorithm. All execution times are averaged over ten runs. The standard deviation was below 4% of the average execution time for all kernels except for the Barnes-Hut tree-building and force-approximation kernels, for which the standard deviation was approximately 10% of the mean running time. This is a direct result of significant changes in the graph layout during the first few iterations that the algorithm runs. This affects Barnes-Hut tree structure, and as such the depth and structure of subsequent Barnes-Hut tree-traversal patterns during force approximation and tree insertions.

Apart from assessing the computation time as we will do in the subsection below, it may be interesting to look at the actual visualizations produced by the algorithms, in particular for the newly created network datasets. Figure 3 shows a visualization of the corporate network CORPNET$_4$, in which some smaller near-clique clusters are visible. These clusters appear to be groups of firms bound together by administrative ties, for example as a result of shared board members between entities of the same firm in different countries. The GitHub collaboration network in Figure 4 shows how there are a number of extremely densely connected groups of repositories. These are likely repositories that are

Table I

PROPERTIES OF THE GIANT COMPONENTS OF THE CONSIDERED NETWORKS (NUMBER OF NODES, EDGES, AVERAGE DEGREE $\overline{deg}$, DENSITY, CLUSTERING COEFFICIENT AND AVERAGE DISTANCE $\bar{d}$), FOLLOWED BY SPEEDUPS FOR THE SIX KERNELS DISCUSSED IN SECTION IV-B. LAST THREE COLUMNS INDICATE THE OVERALL SPEEDUP AND COMPUTATION TIME IN MINUTES ON GPU AND CPU.

Network	**Nodes**	**Edges**	$\overline{deg}$	**Dens.**	**Clus.**	$\overline{d}$	**Grav.**	**Attr.**	**BH-B**	**BH-F**	**Speed**	**Disp.**	**Total**	**GPU**	**CPU**
ca-AstroPh	17,903	196,972	11.0	0.615	0.633	4.35	32×	134×	34×	54×	45×	86×	**50**×	0.015	0.75
ca-CondMat	21,363	91,286	4.3	0.200	0.642	5.51	48×	158×	43×	65×	55×	109×	**58**×	0.016	0.91
cit-HepTh	27,400	352,059	13.0	0.469	0.314	4.44	43×	120×	36×	67×	60×	95×	**61**×	0.020	1.19
email-Enron	33,696	180,811	5.4	0.159	0.509	4.13	52×	121×	42×	48×	67×	110×	**47**×	0.025	1.19
cit-HepPh	34,401	420,828	12.0	0.356	0.286	4.45	48×	131×	47×	55×	66×	102×	**55**×	0.027	1.49
ppi-gcc	37,333	135,618	3.6	0.097	0.075	8.01	67×	110×	44×	52×	77×	124×	**50**×	0.027	1.36
brightkite-edges	56,739	212,945	3.8	0.066	0.173	5.11	88×	146×	72×	73×	111×	154×	**72**×	0.033	2.35
p2p-Gnutella31	62,561	147,877	2.4	0.038	0.005	6.13	117×	205×	105×	89×	145×	195×	**90**×	0.036	3.27
soc-Epinions1	75,877	405,738	5.3	0.070	0.138	4.46	102×	123×	87×	73×	137×	175×	**75**×	0.045	3.41
soc-Slashdot0902	82,168	582,532	7.1	0.086	0.060	4.21	125×	101×	92×	79×	143×	189×	**80**×	0.049	3.97
wave	156,317	1,059,331	6.8	0.043	0.423	23.9	160×	213×	154×	107×	225×	245×	**115**×	0.089	10.28
itdk0304	190,914	607,610	3.2	0.017	0.158	7.45	187×	212×	168×	105×	281×	267×	**113**×	0.109	12.33
gowalla-edges	196,591	950,327	4.8	0.025	0.237	4.87	194×	142×	171×	108×	305×	271×	**115**×	0.116	13.30
m14b	214,765	1,679,018	7.8	0.036	0.425	25.0	187×	213×	175×	113×	310×	275×	**123**×	0.116	14.29
citeseer	220,997	505,327	2.3	0.010	0.101	8.31	178×	221×	154×	91×	294×	248×	**98**×	0.124	12.18
email-EuAll	224,832	340,794	1.5	0.007	0.079	4.27	201×	101×	174×	107×	312×	279×	**114**×	0.127	14.42
web-Stanford	255,265	1,941,926	7.6	0.030	0.619	7.69	160×	101×	158×	92×	270×	237×	**98**×	0.163	16.06
amazon0302	262,111	899,792	3.4	0.013	0.420	9.15	178×	193×	165×	99×	280×	255×	**107**×	0.151	16.18
com-dblp	317,080	1,049,866	3.3	0.010	0.632	7.06	189×	189×	177×	93×	308×	270×	**102**×	0.208	21.16
cnr-2000	325,557	2,738,969	8.4	0.026	0.453	11.0	193×	86×	176×	89×	289×	267×	**96**×	0.233	22.28
web-NotreDame	325,729	1,090,108	3.3	0.010	0.235	7.75	222×	170×	199×	98×	373×	293×	**109**×	0.216	23.43
mathSciNet	332,689	820,644	2.5	0.007	0.410	7.56	208×	266×	195×	103×	328×	292×	**113**×	0.219	24.60
com-amazon	334,863	925,872	2.8	0.008	0.397	12.3	192×	154×	175×	86×	289×	269×	**95**×	0.218	20.75
auto	448,695	3,314,611	7.4	0.016	0.415	37.7	190×	203×	186×	65×	337×	265×	**76**×	0.393	29.80
dblp20080824	511,163	1,871,070	3.7	0.007	0.639	6.66	230×	173×	207×	69×	495×	319×	**79**×	0.517	40.76
web-BerkStan	654,782	6,581,871	10.0	0.015	0.007	7.21	234×	161×	206×	58×	479×	320×	**68**×	0.846	57.37
web-Google	855,802	4,291,352	5.0	0.006	0.055	6.37	240×	252×	203×	48×	480×	312×	**56**×	1.283	72.07
eu-2005	862,664	16,138,468	19.0	0.022	0.602	4.90	239×	156×	207×	56×	520×	320×	**68**×	1.242	84.09
imdb	880,455	37,494,636	43.0	0.048	0.806	4.10	248×	162×	186×	45×	528×	322×	**61**×	1.479	90.04
youTube	1,134,890	2,987,624	2.6	0.002	0.006	4.70	262×	164×	231×	45×	619×	276×	**53**×	1.925	101.7
GitHub	1,271,422	13,045,696	10.0	0.008	0.640	11.7	275×	160×	234×	42×	677×	331×	**52**×	2.342	120.8
in-2004	1,353,703	13,126,172	9.7	0.007	0.574	8.70	257×	166×	198×	42×	589×	327×	**50**×	2.603	129.7
flickr-links	1,624,992	15,476,836	9.5	0.006	0.112	5.19	265×	186×	200×	34×	625×	322×	**40**×	3.901	157.9
as-skitter	1,694,616	11,094,209	6.5	0.004	0.005	5.04	265×	225×	194×	34×	606×	320×	**41**×	3.872	157.7
enwiki-20071018	2,070,367	42,336,614	20.0	0.010	0.104	3.20	266×	85×	199×	34×	603×	322×	**42**×	5.529	232.7
wikipedia	2,388,953	4,656,682	1.9	0.001	0.002	3.50	261×	165×	218×	36×	652×	272×	**42**×	5.417	226.7
CORPNET-3	3,174,496	53,879,276	17.0	0.005	0.533	6.54	255×	120×	232×	37×	662×	272×	**46**×	8.067	368.3
CORPNET-4	4,602,225	123,329,543	27.0	0.006	0.713	6.90	246×	55×	224×	33×	637×	250×	**40**×	13.83	545.8

edited by bots or automated editors, as it is highly unlikely that developers contribute to so many different repositories. Indeed, the visualizations attain the goal of visualization: observing patterns and outliers in the data.

D. Discussion

Table I lists how most components of the force-directed layout algorithm are able to attain speedups of 200× or more, which is in line with theoretical expectations with respect to the parallelizable aspects of these algorithms and the parallel capabilities of GPUs in general. Furthermore, the overall speedups between 40× and 123× indicate that running graph layout algorithms on the GPU can truly help overcome the computational challenges of large-scale network visualization. In particular, consider that for the largest datasets containing millions of nodes, at the bottom of the table, the execution time is reduced from hundreds of minutes (i.e., hours) to at most 14 minutes. So, by means of a relatively straightforward implementation of the algorithm on the GPU, work on the visualization of large-scale networks can be made practically possible.

Real-world networks share a number of interesting properties, as discussed in Section II-A. Interesting to note is that regardless of the size of the network, large speedups are attained. Note that the speedup also does not appear to be affected by core network characteristics such as the average degree, distance and clustering coefficient, indicating that the method is suitable for a diverse range of real-world networks. Furthermore, the components handling gravity, speed and displacement show higher speedups for larger networks, indicating that full advantage of the GPU is being taken as more data is used.

Attractive force calculation (column "Attr" in Table I), despite avoiding thread-divergence, requires the use of atomic operations to update force attributes of neighboring nodes. Since we sort edges by source node for coalesced access (see Section IV-B), multiple edges for the same source node are processed concurrently on a single multiprocessor. This

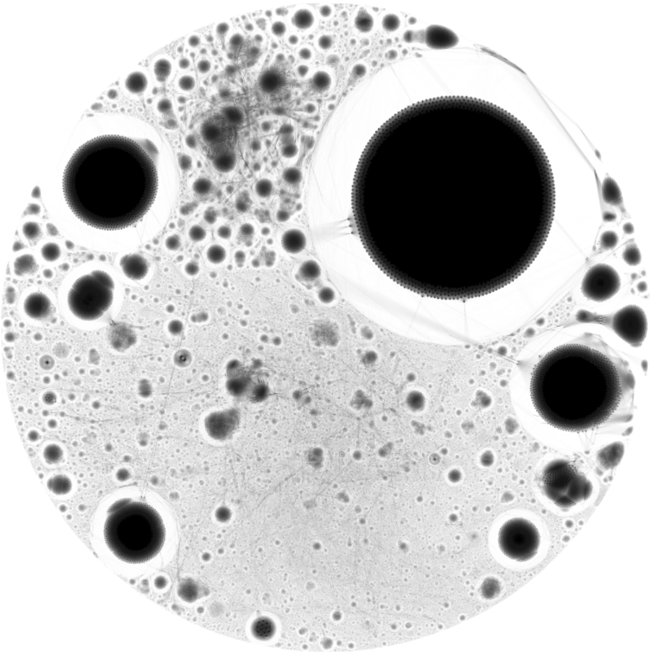

Figure 4. Resulting layout of the GITHUB network with $1,271,422$ nodes and $13,045,969$ edges.

allows for coalesced memory reads from nodes' properties, but it will also cause threads to face blocking memory writes as they try to atomically update force values for identical nodes. An increased average node degree appears to affect this conflict, as can for example be seen from the results for the CORPNET$_3$ and CORPNET$_4$ datasets. The latter has a larger average node degree, resulting in a significantly lower speedup. Similarly, for datasets enwiki-20071018 and cnr-2000 the speedups are limited, which may again be a result of the relatively large size of the network combined with relatively high average node degree values.

Whereas the gravity, speed and displacement components show speedups well beyond a factor 200 for the larger datasets, the overall speedup is clearly lower. This appears to be caused by the performance of the two Barnes-Hut kernels in the *BodyRepulsion* component, denoted by columns "BH-B" (tree building) and "BH-F" (repulsive force approximation). The force approximation in particular appears to be constrained to a speedup of approximately a factor 45, which is relatively low compared to most other kernels that show speedups at least four times as large. This discrepancy in speedup likely results from the irregular memory access patterns associated with the tree traversals made both during force approximation and tree construction. Considering that the body repulsion kernels constitute approximately 80% of the execution time, repulsive force computation is currently the bottleneck in the pipeline and constrains the overall speedup of the computation. Yet, it should be noted that this constraint on overall speedup does not appear to be directly related to the size of the network, which indicates that the proposed implementation is more than sufficiently scalable.

VI. CONCLUSION

In this paper, we have proposed a framework to implement graph visualization algorithms on the GPU. We have demonstrated how the popular ForceAtlas2 algorithm is implemented within this framework by means of a number of specific components. Evaluation of the performance of the resulting implementation showed that significant speedups are attained. This is the case for the different components as well as the GPU algorithm as a whole. For the largest networks, the computation time to produce a high-quality visualization is reduced from 9 hours to only 14 minutes. Given the similarity between force-directed algorithms, in that they mostly differ in their force-model and choice of adaptive speed, we expect our findings to generalize to other force-directed algorithms as well.

The speedups attained as a result of this research are of significance to the network analysis community, where visualization and manual inspection of patterns and outliers in networks is a common activity. A particularly interesting outcome is the fact that the algorithm performs well across a range of diverse datasets of different origins. This suggests that the proposed implementation is sufficiently generic to handle visualization of virtually any type of real-world network. Furthermore, the performance (as expected) appears to scale linearly with the number of nodes, which is important given the sheer size of typically considered network data.

The framework component to compute repulsive forces between nodes was found to be the main limitation from achieving better overall performance. Indeed, the Barnes-Hut implementation used in this work limits the overall speedup we can achieve, which is why evaluating alternatives for this step would be a logical step for future research. Furthermore, the performance of both the baseline CPU code and the GPU code can be further improved, resulting in in-depth performance analyses of optimized and tuned implementations. Considering that force approximation constitutes a significant share of the execution time of the graph layout algorithm, performance improvements to this component will also directly result in higher overall speedups.

Furthermore, the performance improvements reported in this paper pave the way towards interactive representations of the graph layouts. For graphs with a few hundred thousand nodes, the time needed to produce a layout has been reduced from tens of minutes to a fraction of a minute. All in all, this means that using GPUs, it is feasible to fully re-layout a graph in response to interactive user input.

ACKNOWLEDGMENTS

The third author was supported by funding from the European Research Council (ERC) under the European

Union's Horizon 2020 research and innovation programme (grant agreement number 638946).

REFERENCES

[1] H. Gibson, J. Faith, and P. Vickers, "A survey of two-dimensional graph layout techniques for information visualisation," *Information Visualization*, vol. 12, no. 3-4, pp. 324–357, 2013.

[2] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: An annotated bibliography," *Computational Geometry*, vol. 4, no. 5, pp. 235 – 282, 1994.

[3] Y. Hu and L. Shi, "Visualizing large graphs," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 2, pp. 115–136, 2015.

[4] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian, "Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software," *PLoS ONE*, vol. 9, no. 6, pp. 1–12, 2014.

[5] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks." in *Proceedings of International Conference on Web and Social Media (ICWSM)*, 2009, pp. 361–362.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[7] G. H. Dal, W. A. Kosters, and F. W. Takes, "Fast diameter computation of large sparse graphs using GPUs," in *Proceedings of 22nd International Conference on Parallel and Distributed Processing (PDP)*, 2014, pp. 632–639.

[8] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 185–195.

[9] A. Godiyal, J. Hoberock, M. Garland, and J. C. Hart, "Graph drawing," I. G. Tollis and M. Patrignani, Eds., 2009, ch. Rapid Multipole Graph Drawing on the GPU, pp. 90–101.

[10] Y. Frishman and A. Tal, "Multi-level graph layout on the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1310–1319, 2007.

[11] S. Ingram, T. Munzner, and M. Olano, "Glimmer: Multilevel mds on the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 2, pp. 249–261, Mar. 2009.

[12] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, 2000, pp. 163–170.

[13] The Khronos Group Inc., "The open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl/, accessed: 02-10-2016.

[14] NVIDIA, "CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/, accessed: 02-10-2016.

[15] ——, "CUDA C Best Practices Guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2015, accessed: 20-08-2016.

[16] D. E. Knuth, "Computer-drawn flowcharts," *Communications of the ACM*, vol. 6, no. 9, pp. 555–563, 1963.

[17] W. Tutte, "How to draw a graph," *Proceedings of the London Mathematical Society*, vol. 13, no. 1, pp. 743–767, 1963.

[18] S. G. Kobourov, *Handbook of Graph Drawing and Visualization*. Chapman and Hall/CRC, 2013, ch. Force-Directed Drawing Algorithms, pp. 383–408.

[19] R. Tamassia, *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007.

[20] P. A. Eades, "A heuristic for graph drawing." in *Congressus Numerantium*, vol. 42, 1984, pp. 149–160.

[21] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[22] A. Frick, A. Ludwig, and H. Mehldau, "A fast adaptive layout algorithm for undirected graphs," in *Proceedings of the International Workshop on Graph Drawing*, 1995, pp. 388–403.

[23] R. Davidson and D. Harel, "Drawing graphs nicely using simulated annealing," *ACM Transactions on Graphics*, vol. 15, no. 4, pp. 301–331, 1996.

[24] S. Aluru, J. Gustafson, G. Prabhu, and F. E. Sevilgen, "Distribution-independent hierarchical algorithms for the n-body problem," *Journal of Supercomputing*, vol. 12, no. 4, pp. 303–323, 1998.

[25] S. Hachul and M. Jünger, "Drawing large graphs with a potential-field-based multilevel algorithm," in *Proceedings of the 12th International Symposium on Graph Drawing*, 2005, pp. 285–295.

[26] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[27] NVIDIA, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015, accessed: 20-08-2016.

[28] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm," in *GPU Computing Gems Emerald Edition*, W. mei W. Hwu, Ed., 2011, ch. 6, pp. 75–92.

[29] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proceedings WWW*, 2013, pp. 1343–1350.

[30] F. W. Takes and E. M. Heemskerk, "Centrality in the global network of corporate control," *Social Network Analysis and Mining*, vol. 6, no. 1, pp. 1–18, 2016.