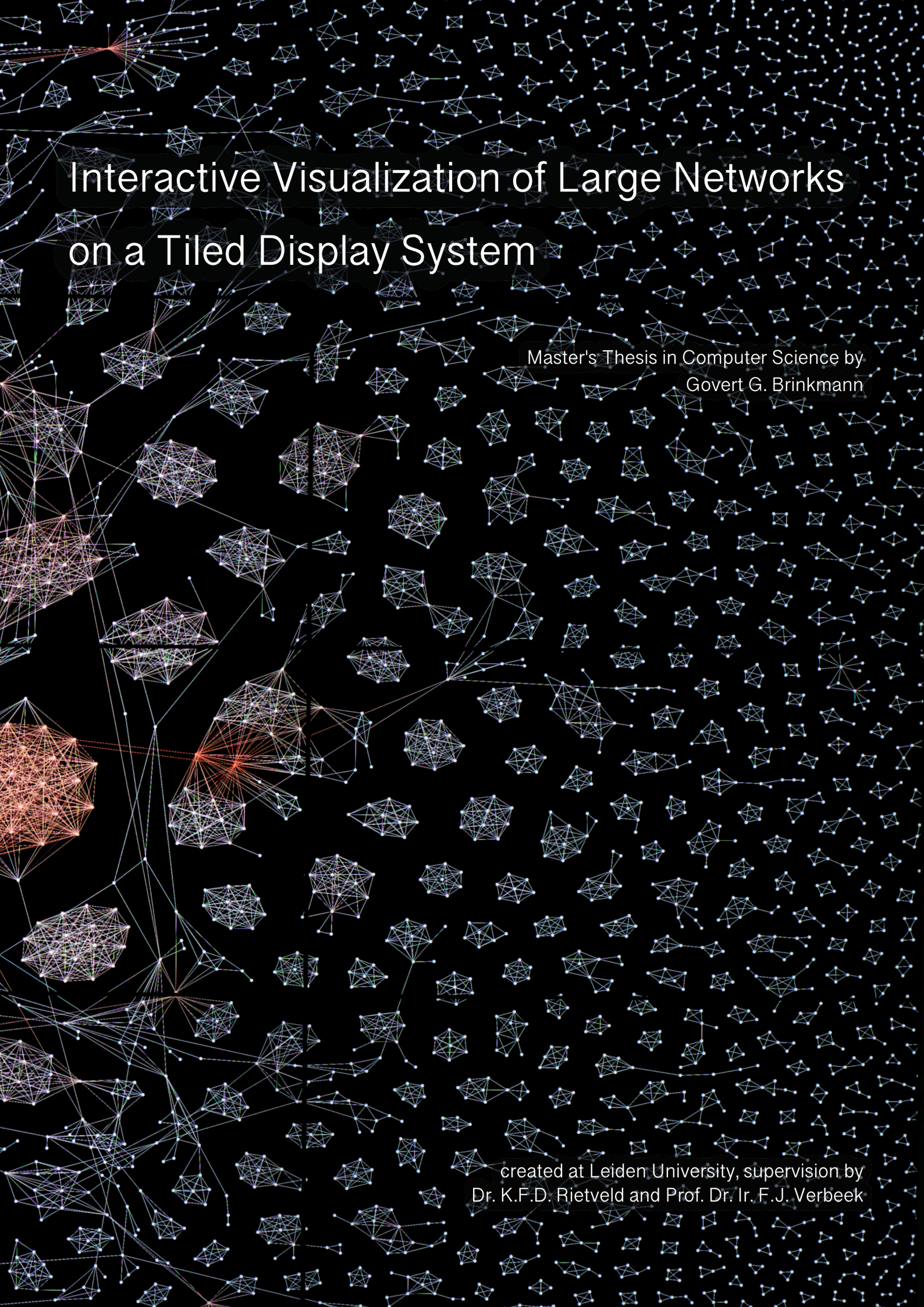


Interactive Visualization of Large Networks on a Tiled Display System

Master's Thesis in Computer Science by
Govert G. Brinkmann

created at Leiden University, supervision by
Dr. K.F.D. Rietveld and Prof. Dr. Ir. F.J. Verbeek

Universiteit Leiden
Opleiding Informatica

Interactive Visualization of Large Networks

on a Tiled Display System

Name:	G. G. Brinkmann
Date:	31/07/2018
Supervisors:	Dr. K.F.D. Rietveld Prof. Dr. Ir. F.J. Verbeek

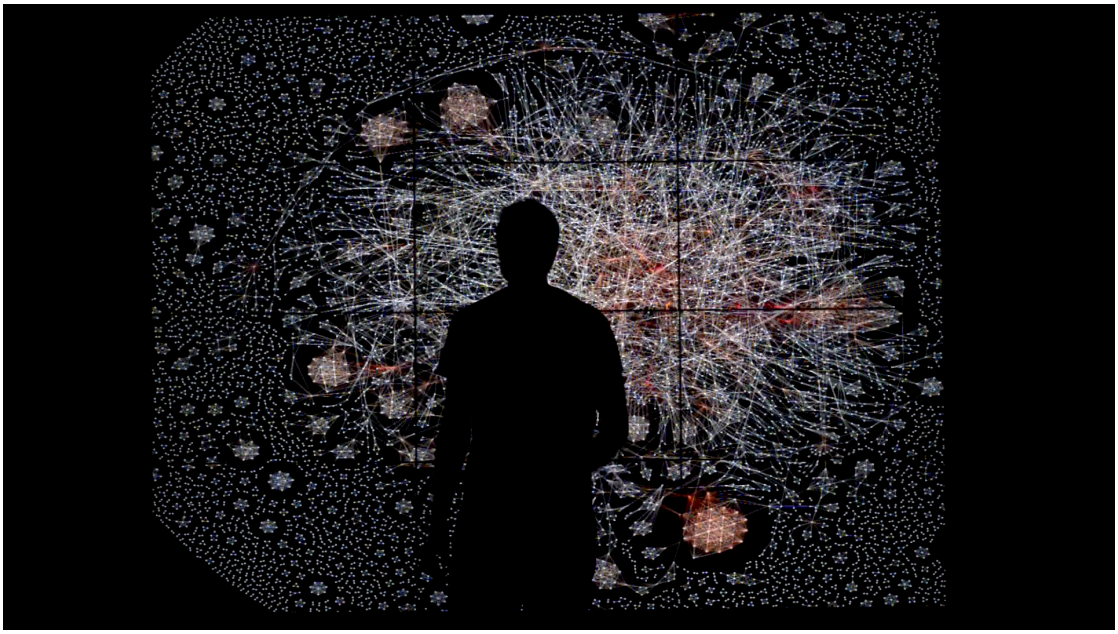
MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Advances in the design and implementation of network drawing algorithms enabled the visualization of large networks with millions of nodes and edges. Given the prevalence of network data, this enables insights in problems crossing many fields of study. Unfortunately, viewing the network drawings for large networks on a typical desktop monitor remains challenging, due to its limited amount of screen space. As such, the use of tiled display systems, composed of multiple monitors arranged as tiles, has been suggested for network visualization. In this thesis we extend research on network visualization using tiled display systems, by considering how the graphics processing units (GPUs) in a tiled display system can be used to both compute network layouts and to subsequently render network drawings, at interactive frame rates. A recent GPU-based implementation of a force-directed graph layout algorithm is extended to utilize the processing power of multiple GPUs, and it is combined with a distributed rendering approach in which each graphics card in the tiled display system renders the part of the network to be displayed on the monitors attached to it. To the best of our knowledge, this is the first report discussing a multi-GPU network visualization approach. The multi-GPU implementation of the force-directed layout algorithm we present, which is based on distributed repulsive force approximation using quadtrees, scales moderately with increasing numbers of GPUs. The multi-GPU network renderer scales better, but its performance is reduced due to uneven load balancing. However, our evaluation of the approach on a single-node, 25 megapixel tiled display system with three GPUs, demonstrated interactive performance at 60 frames per second for real-world networks with tens of thousands of nodes and edges. This constitutes a performance improvement of approximately 3.9× over our initial single GPU implementation, that uses operating system support to span drawings across all monitors in the tiled display system.

Keywords: Network Visualization, Tiled Display Systems, Interactive Visualization, Computer Systems, GPU, CUDA, OpenGL



Contents

Unit Prefixes

Unit prefixes from the International System of Units (SI) are used throughout this thesis. As such 1 KB corresponds to 1000 bytes. If binary prefixes are used, they are according to ISO/IEC 80000:2008. Hence, 1KiB corresponds to 1024 bytes.

Additional Resources

Additional information, videos and the source code for all programs developed for this study can be obtained via `https://govertbrinkmann.nl/mthesis`.

Chapter 1

Introduction

In this study we explore real-time interactive network visualization on a tiled display system, using its graphics processing units (GPUs). Our focus is on a 25 megapixel non-distributed tiled display system, composed of twelve monitors connected to a single machine, and on large networks with tens of thousands of nodes and edges.

A network (or graph) describes relationships between a set of entities, and visualization is one of the primary techniques people use to analyze these relationships. We consider the most common visualization method, in which the network's entities (nodes) are drawn as dots in the plane, with lines (edges) connecting any two related nodes. Figure 1.1 provides an example, a drawing of a small protein-protein interaction network.

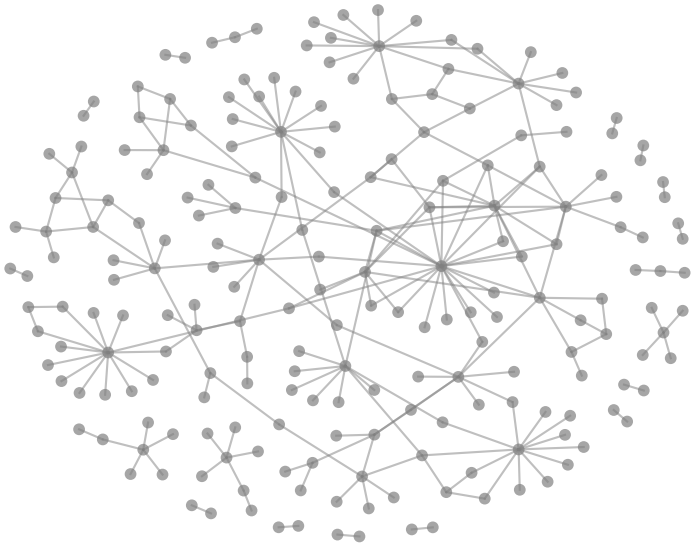


Figure 1.1: Drawing of a protein-protein interaction network [4, 32]. Nodes represent proteins, edges the interactions between them.

The challenge in drawing a network is positioning the nodes in such a way that the resulting layout clearly depicts the structure of the network. After all, there are (infinitely) many ways to draw the same network. In general, one wants to position a given node in proximity of related nodes, and at distance of unrelated nodes. Also, edge crossings, overlapping nodes and long edges should be prevented as much as possible. For humans, this becomes difficult when the size of the network exceeds a dozen of edges, which led to the development of graph layout algorithms. Although the first graph layout algorithms were designed to operate on thousands of nodes, recent algorithms scale to millions of nodes and edges [26]. Besides being a result of algorithmic improvements, this is also due to improved implementations.

However, visualizing very large networks on a typical desktop computer still poses challenges. Due to the relatively small amount of screen space that is available, viewing the layout of a very large network, such as the one depicted in Figure 1.2, results in excessive visual clutter. An interactive presentation that allows for zooming and panning can overcome this issue, however this reduces the number of nodes and edges that can be viewed at once, and thereby the viewer's ability to grasp the overall structure of the network. As such, applying the standard *'overview first, zoom and filter, then details on demand'* information seeking mantra remains challenging for visualizations of large networks [51].

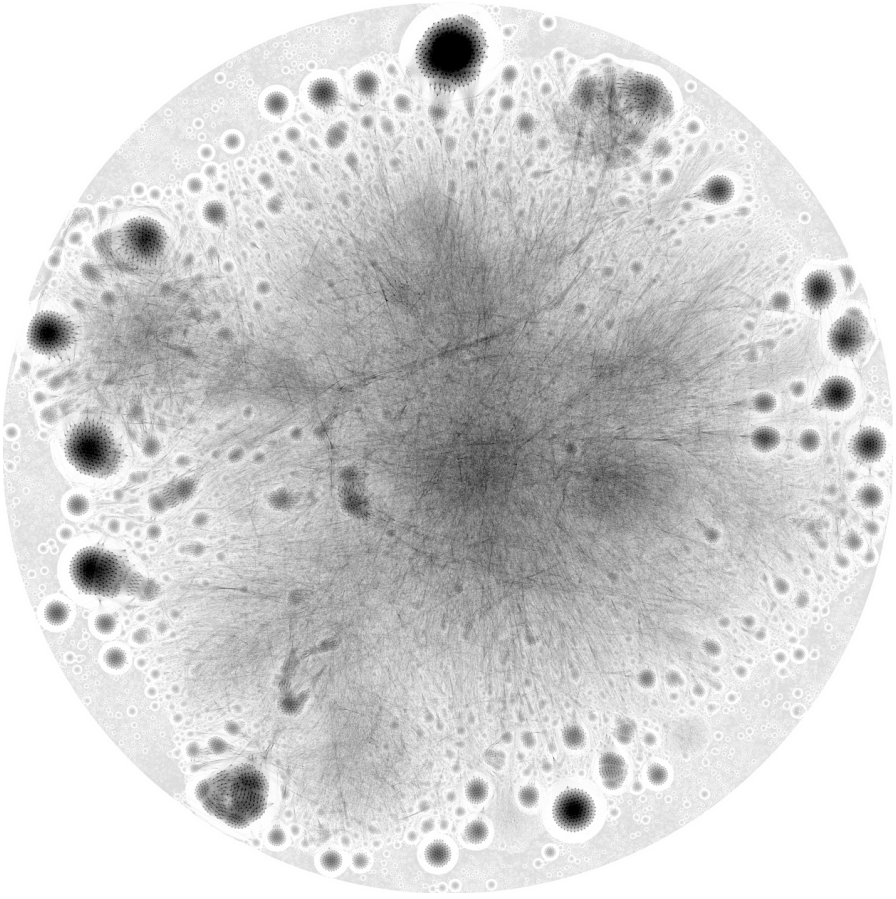


Figure 1.2: Network drawing for a large network with 391,966 nodes and 1,711,968 edges. Nodes represent companies, which are connected if their boards of directors share a member [52].

To overcome this problem, the use of tiled display systems has been suggested [10, 37]. Tiled display systems are composed of multiple displays, usually identical models, which are arranged as tiles to form a single large display area. For an example, see Figure 1.3. Tiled display systems provide a scalable and cost-effective solution to the limited amount of screen space provided by a typical desktop computer. Depending on the number of displays, their resolution, and the demands of the application, the displays connect to a single computer or a distributed cluster consisting of multiple computers. In this thesis we focus on the former case, in which all monitors connect to a single machine.

Figure 1.3: 'BigEye', the tiled display system we consider for this study.

To the best of our knowledge, earlier studies on the use of tiled display systems for network visualization relied on the system's central processing units (CPUs) to compute network layouts. For the present study we consider using the system's graphics processing units (GPUs) instead. Tiled display systems generally contain multiple GPUs, located on the graphics cards that connect the displays to the system, which provide tremendous computational power. It has been shown that the GPU allows for high-performance implementations of graph layout algorithms [8, 20, 36]. Therefore we hypothesize that using the GPUs in a tiled display system allows for high-performance visualization of large-scale networks using its full resolution. More specifically, we expect to achieve real-time performance, where user's input is accounted for without visible delay, for networks with hundreds of thousands of nodes and edges. Real-time performance combined with interactivity improves the user's ability to effectively analyze a network. Besides easing data exploration, it allows users to manually steer the layout process, which can be crucial to prevent the layout algorithm from converging to a sub-optimal layout.

To test our hypothesis, we derive a multi-GPU implementation of ForceAtlas2 [30], a commonly used algorithm for network visualization. This implementation is combined with a distributed rendering approach in which each GPU renders the part of the network to be displayed on the monitors attached to it. To assess the resulting system's interactive capabilities, we will implement a number of interactions using a wireless control device that enables natural interactions with visualizations on the wall-sized display. However, before confronting the network visualization problem we first implement an interactive visualization of the Mandelbrot set on the tiled display system. This preliminary exploratory study allows for the development of a 'framework' implementing the aforementioned visualization approach, which involves distributed general purpose computations and distributed rendering using the GPUs in the system. The preliminary study also enables us to discover any technical challenges related to the specific tiled display system we focus on for this study.

The content of this thesis is structured as follows. In Chapter 2 we first discuss the methods and materials used for this study. As such, we discuss graph layout algorithms, tiled display systems, the GPU as a platform for general purpose computation, as well as definitions for concepts used throughout the thesis. We also discuss a number of existing approaches to network visualization on tiled display systems, and we detail BigEye, the tiled display used in this study. The remainder of the thesis consist of two parts. In Part I we present the preliminary exploratory study on visualizing the Mandelbrot set. The 'framework' resulting from this forms the starting point for Part II of the thesis, where we apply it for interactive network visualization. Chapters 7 and 8 present a discussion of our results and our conclusion, respectively.

Chapter 2

Methods and Materials

In this chapter, we review the methods and materials related to our study on network visualization using tiled display systems. For clarity, we also define a number of commonly used concepts. As such, we first define what a network, or graph, is. In the next section, we introduce graph layout algorithms in general, and force-directed graph layout algorithms in specific, since the latter will be used extensively in the remainder of this thesis. Next, we describe tiled display systems, common designs and their use for network visualization. We also discuss the architecture of graphics processing units (GPUs), and how they enable high-performance implementations of certain general purpose computations, given that this is a method which is central to the approach taken in this thesis. We conclude the chapter by introducing BigEye, the tiled display system used for this study.

2.1 Graphs

We define a *graph* $G = (V, E)$, as a set of *vertices*, or *nodes*, together with a set of *edges*, or *links*. The set of edges, denoted E, represents a (binary) relationship on the set of vertices, denoted V, such that for *directed* graphs $E \subseteq V \times V$, and for *undirected* graphs $E = \{\{u, v\} : u, v \in V\}$. Both for directed and undirected graphs, we denote the existence of an edge between nodes u and v by $(u, v) \in E$ and we do not consider networks with self-loops, i.e. edges $(u, v) \in E$ s.t. $u = v$. Instead of $(u, v) \in E$, we can also write that node v is *adjacent* to node u. If $(u, v) \in E$ or $(v, u) \in E$, then nodes v and u are *neighbors*.

The *density* of a graph is defined as the ratio between the number of edges and the maximum number of edges, given the number of nodes. For directed graphs the density thus equals $|E|/(|V|(|V| - 1))$, whereas it equals $|E|/(\frac{1}{2}|V|(|V| - 1))$ for undirected graphs. The *out-degree* of node u is defined as the number of nodes adjacent to it, i.e. it equals $|\{v \in V : (u, v) \in E\}|$. Similarly, the *in-degree* of a node v equals the number of nodes to which it is adjacent, i.e. it equals $|\{u \in V : (u, v) \in E\}|$. Note that for undirected graphs the out-degree of a node $v \in V$ equals its in-degree, and as such we can refer to both as the *degree* of this node, denoted $deg(v)$.

Two nodes $u, v \in V$ are *connected* if a *path* between them exists. A length-n path between nodes u and v consists of a sequence of nodes $(x_0, x_1, \ldots, x_n)$ such that $x_0 = u$, $x_n = v$ and $\forall i, 0 \leq i < n : (x_i, x_{i+1}) \in E$. The *shortest path* between nodes u and v is defined as the path with the shortest length that connects nodes u and v, and its length is referred to as the *distance* between these nodes, denoted $d(u, v)$. If no path between u and v exists, i.e. they are not connected, $d(u, v) = \infty$. A graph is *strongly connected* if all node-pairs are connected. If

all node-pairs are connected when disregarding edge direction, i.e. when assuming $(u, v) \in E \Rightarrow (v, u) \in E$, a graph is *weakly connected*.

Given a graph $G = (V, E)$, we define the subgraph induced by the set of nodes $V' \subseteq V$ to be the graph $G' = (V', E')$, such that $(u, v) \in E' \Leftrightarrow u, v \in V' \wedge (u, v) \in E$. The *weakly connected components* (WCCs) of a graph are defined to be its maximal weakly connected subgraphs. Similarly, the *strongly connected components* (SCCs) of a graph are defined to be its maximal strongly connected subgraphs. These components are maximal in the sense that no nodes can be added without voiding the connectivity property. The WCCs of a graph are sometimes also referred to as the *components* of a graph. We focus in particular on the largest weakly connected component (LWCC) of a network.

In this thesis we do not make a distinction between networks and graphs, but use the terms interchangeably.

2.2 Graph Layout Algorithms

To reveal the structure of a graph, it is commonly drawn in the plane with vertices represented as points, and edges as straight lines connecting related vertices. As discussed in the introduction, the challenge in drawing a graph is assigning a position $\mathbf{p}_v \in \mathbb{R}^2$ to each $v \in V$ such that a 'readable' layout emerges. In a readable layout related nodes are generally positioned in spatial proximity of each other, whereas unrelated nodes are at distance of each other. Also, overlapping nodes and edge crossings should be avoided. Most importantly, the layout should reflect the structure of the network. The latter is illustrated by Figure 2.1 which shows that different layouts for the same network can suggest structural differences between the network that is represented. The left layout (correctly) reflects uniformity between the nodes, whereas node zero is (incorrectly) emphasized in the right layout, due to its position at the center of the drawing.

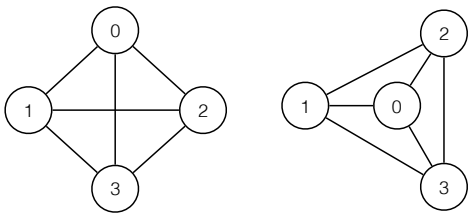


Figure 2.1: Two drawings of the same (fully connected) network.

Figure 2.1 also demonstrates how criteria for a readable layout can be in conflict with each other. Although the left drawing reflects the topological uniformity between nodes, it does not avoid edge crossings.

Whilst determining readable layouts for networks with less than a dozen of edges is feasible for a human, it is desirable to use a computer and a graph layout algorithm as the network size increases. A graph layout algorithm takes as input the graph $G = (V, E)$, potentially with additional information on nodes and edges, and computes for each $v \in V$ a position $\mathbf{p}_v \in \mathbb{R}^2$, such that a readable layout emerges. From the different types of graph layout algorithms that have been conceived [26, 19], we choose to focus on force-directed graph layout algorithms. As

the remainder of this section will describe, force-directed layout algorithms are especially suitable to the interactive applications this thesis considers.

The force-directed approach to graph layout, considers the layout of a graph to be a physical system in which nodes interact as physical bodies. The force model describing these interactions is chosen such that the evolution of the system over time causes a readable layout to emerge. As such, an attractive force between neighboring nodes generally serves to move related nodes towards each other. In contrast, a repulsive force moves all node-pairs away from each other to place unrelated nodes at a distance of each other. A gravitational force that moves nodes towards the center of the layout can be introduced, to ensure the different components of the graph stay in proximity of each other. This force model is also referred to as the spring-electic model, given that it corresponds to a system with charged particles, i.e. the nodes, connected by springs, i.e. the edges.

Algorithm 1 formalizes the force-directed approach to graph layout. The values used for algorithms' parameters are generally chosen through trial-and-error, increasing the number of iterations (it_{max}) with the network size.

Algorithm 1 Force-Directed Graph Layout

Input: Graph $G = (V, E)$, it_{max} (number of layout iterations), k_g (gravitational force scalar), k_a (attractive force scalar) k_r (repulsive force scalar), δ_t (time step size).
Output: For each $v \in V$, a position $\mathbf{p}_v \in \mathbb{R}^2$.

for all $v \in V$ **do** ▷ Randomize layout
  $\mathbf{p}_v \leftarrow \text{RANDOM}()$
end for
▷ Start layout process
for $i = 1 \rightarrow it_{max}$ **do**
  for all $v \in V$ **do**
    $\mathbf{f}_v \leftarrow -\frac{k_g}{|\mathbf{p}_v|^2} * \mathbf{p}_v$ ▷ Gravity
    for all $w \in \text{NEIGHBORS}(v)$ **do**
      $\mathbf{f}_v \leftarrow \mathbf{f}_v + k_a * \frac{\mathbf{p}_w - \mathbf{p}_v}{|\mathbf{p}_w - \mathbf{p}_v|}$ ▷ Attraction
    end for
    for all $w \in V, w \neq v$ **do**
      $\mathbf{f}_v \leftarrow \mathbf{f}_v + k_r * \frac{\mathbf{p}_v - \mathbf{p}_w}{|\mathbf{p}_v - \mathbf{p}_w|}$ ▷ Repulsion
    end for
  end for
  for all $v \in V$ **do**
    $\mathbf{p}_v \leftarrow \mathbf{p}_v + \delta_t * \mathbf{f}_v$ ▷ Displacement
  end for
end for

Force-directed layout algorithms are well suited to interactive applications for multiple reasons. First, their resemblance to physical systems connects to users their intuition. Also, the iterative approach, which corresponds to a time-continuous process, allows for the entire layout process to be visualized. As such users can comprehend the layout procedure, and better understand the effect the different parameters have. The force-directed method also accounts for interaction with the layout process, allowing users to manually steer it to avoid the layout from converging to a local minimum.

2.3 Graphics Processing Units

For the present study we use graphics processing units (GPUs) for network visualization, i.e. to implement the layout algorithm and the network renderer that generates network drawing from the layouts. In contrast to the CPU architecture, that is optimized to achieve low latency on a wide range of computational problems, the GPU is optimized more to achieve high throughput on (highly) data-parallel problems [16]. This is partly due to the relatively great number of functional units that the GPU features, which generally come at the expense of omitting the advanced, latency-reducing, features employed by the CPU such as branch prediction and speculative execution. Although the design of the GPU has its origins in computer graphics applications, which involve the processing of many independent pixels and geometrical primitives, the architecture has increasingly been applied to achieve high-performance implementations for general purpose computations [5]. Whereas general purpose computation on GPUs (GPGPU) initially required formulating problems using graphics APIs such as OpenGL [50], dedicated GPGPU frameworks such as CUDA [43] and OpenCL [22] have since been released.

Given the large number of independent, per-node, operations used in force-directed graph layout algorithms, the GPU has enabled high-performance implementations [8, 21, 14] of these algorithms as well. For the present study we implement the network layout algorithm using CUDA and the network renderer using OpenGL. The CUDA-OpenGL interoperability API [46] is used to share the data structures representing the network layout between CUDA and OpenGL. This approach, which differs from the initial studies on network visualization using GPUs [14], allows both parts of the visualization process, layout and rendering, to be represented naturally, whilst avoiding unnecessary data transfers between CPU and GPU memory.

NVIDIA GPUs and CUDA

Since we use the NVIDIA compute unified device architecture (CUDA) platform [43] to implement our multi-GPU network layout algorithm, which is discussed in Section 5.3, we use this section to provide some background information on CUDA and the architecture of NVIDIA GPUs.

CUDA enables programmers to implement general purpose computations on the highly parallel architecture provided by NVIDIA GPUs. For our study we use CUDA C, which is an extension to C programming language. Using CUDA C, programmers can specify *kernels*, which are functions to be executed in parallel on the GPU using many *threads* of execution. Although threads execute the same kernel (function), they generally operate on different data elements, thus parallelizing a computation. CUDA threads are organized by the programmer into a *grid* of *thread blocks*. When launching a grid for execution, the blocks contained therein are assigned to the different *multiprocessors* (MPs) on the GPU, which subsequently subdivides the thread block into *warps* of 32 threads. The threads in a warp concurrently execute the same instruction, through a *single instruction multiple threads* (SIMT) architecture and the MP is equipped with dedicated hardware to interleave the execution of different warps in order to hide latency. To achieve optimal hardware utilization, it is important that the threads in a warp share execution paths, since diverged threads in a warp cannot execute concurrently. Also, to effectively utilize the memory bandwidth provided by the GPU, the threads in a warp should access consecutive ranges of aligned memory. This allows for multiple memory accesses to be *coalesced* into single transactions, which greatly improves memory bandwidth utilization.

2.4 Network Visualization using Tiled Display Systems

Tiled display systems, composed of multiple monitors arranged as tiles to form a large display area, are often a scalable and cost-effective approach to achieve wall-sized high-resolution displays. Displays can be added and removed depending on the desired display dimensions and resolution, and reusable 'off-the-shelf' components can be used for the system's construction. The monitors in the system can connect to either a single computer or a dedicated computer cluster depending on the number of displays, their resolution and the intended application of the system. The former design, involving only a single computer, might also be considered a 'multi-head' setup in the context of desktop workstations. Still, the latter does not necessarily involve the use of multiple graphics cards, and the resolutions common to the wall-sized displays we focus on in this study. Although this study focuses on the single-node tiled display system available to us, we believe our results extend to cluster based tiled display systems as well.

The use of tiled display systems for network visualization has been proposed to reduce the visual clutter that results from visualizing large networks on a typical desktop monitor. Although interactive systems that allow users to navigate the visualizations of large networks can partially solve this problem, this reduces the possibility to reveal structures at a more global level. Besides, the large display area provided by tiled display systems facilitates collaborative research between multiple people. We found previous research on the use of tiled display systems for network visualization by Mueller et. al. (2006), Chae (2013), Jingai et.al. (2015) and Gu et. al. (2015). In this section we briefly discuss these studies, and compare them to the approach taken for the present study.

Mueller et. al. [37] present a network visualization approach for distributed (tiled display) systems, that uses MPI [13] for parallel computation and Chromium [27] for distributed rendering and display. A distributed force-directed graph layout algorithm is derived, based on the classical algorithm by Fruchterman and Reingold [15], by considering how different approaches to data- and work-distribution affect performance and the layout quality. Realizing performance levels suitable for interactivity is a clear objective of the authors. The authors evaluate their system on an eight monitor tiled display system, connected to an eight node cluster, both in terms of its scalability when using increasing numbers of processors and in terms of the 'costs' resulting from displaying generated layouts. Different types of randomly generated networks are used for the evaluation. For networks with 2000 nodes the results show an average performance improvement of 10× when scaling from one to four processors, after which performance saturates. Using eight processors allowed for randomly generated networks with 8000 nodes, and up to (approximately) $80,000$ edges, to be visualized at frame rates between 2 to 5 frames per second.

Chae [10, 11] presents distributed algorithms for network visualization, which are evaluated on a 200 megapixel cluster-based tiled display system composed of 50 monitors. In the development of the algorithms, techniques are considered to reduce the number of edges crossing between nodes in the cluster, but also to prevent nodes in the layout from being positioned on the bezels of monitors. A modified k-means clustering algorithm is evaluated in order to reduce the running time of the layout algorithm. The algorithms are evaluated both in terms of their scalability and the layout qualities that result from them.

Jingai et. al. [31] report their intermediary results on adapting the open source Gephi [2] network analysis and visualization software to run on a cluster-based tiled display system via a commercial middleware. Since the middleware supports the OpenGL graphics library, that Gephi uses for network rendering, few changes to Gephi were required. The authors report their system was successful at visualizing a protein-protein interaction network with 2361 nodes and 7182 edges, with all labels visible, and that the performance of the system should be assessed in future studies.

Gu et. al. [23] present an approach for the interactive visual analysis of image collections by means of a compound 'iGraph' representing the relationships between the images and keywords in the collection. To visualize the iGraph, a force-directed graph layout algorithm based on the classical algorithm by Fruchterman and Reingold [15] is first used to compute a layout for the backbone of iGraph, which consists of images and keywords that are representative for the collection. This layout is then further refined 'on demand' as the user navigates around the iGraph. The system allows for the interactive filtering and comparison of data in the iGraph, and uses an approach akin to collaborative filtering to recommend interesting data to users for further exploration. The approach is evaluated using two datasets on an eight node cluster using graphs with thousands of nodes and millions of edges. Besides, a method for using the cluster to visualize results on a 50 megapixel tiled display system is discussed. The GPUs in the system are used for data pre-processing, whereas the CPUs in the system are used for the graph layout process. Finally, a user evaluation is discussed.

Similar to the studies discussed in this section, we focus on using force-directed graph layout algorithms for our visualization system. However, we do not consider significant modifications to the algorithms to improve their operation in the context of the tiled display system. We rather focus on improvements in their implementation. Most importantly, we consider using the GPUs in the tiled display system for their implementation rather than the CPU. Also, we do not focus on a distributed tiled display system, but on a single machine with 12 monitors connected to three graphics cards. We consider the multiple GPUs for improved processing power only, without distributing the data between them for increased an memory capacity. Our previous work [8] does not suggest that the memory capacity of current systems bounds the size of networks that can be visualized interactively. In contrast to the studies we discussed in this section, we use a wider collection of large real-world networks for the evaluation of our system. Similar to the study of Mueller et. al., we focus on realizing an interactive system. However, we aim to avoid any perceivable delay between user input and corresponding updates on on the monitors, since this would degrade the extent to which users experience interactivity. We choose to focus on obtaining a frame rate of 60 Hz., matching the refresh rate of the monitors in the tiled display system.

2.5 BigEye

The 'BigEye' tiled display system we used for this study, which is depicted in Figure 1.3, consists of twelve 47" Philips BDL4777XL monitors with a resolution of 1920×1080 pixels, that are arranged in a 3×4 grid. The four monitors in each column are connected to a single NVIDIA GeForce GTX660 graphics card via DVI-I, DVI-D, HDMI and DisplayPort connections. Each graphics card is equipped with 1999MiB of GDDR5 memory. An MSI Big Bang-Marshal (MS-7670) motherboard hosts the three graphics cards, which are connected to it using the PCI Express v2 bus, with 8 PCIe lanes available to each card. Note that the graphics cards are not interconnected via other means, such as an NVIDIA SLI bridge.

Table 2.1 presents some relevant information on the NVIDIA GeForce GTX660 GPU. The peak number of single precision floating point operations per second (peak FLOP/s), corresponds to the throughput achieved when executing fused multiply-add (FMA) instructions on all CUDA cores, at the base clock-frequency. As such it is computed as the product of the number of CUDA cores and the base clock-frequency, times two.

The system is equipped with an Intel Core i7-2600K CPU and 16GiB of DDR3 (1333 MHz) memory. Table 2.2 provides some details on the CPU.

The system runs Ubuntu 16.04.3 LTS, which is configured to use the Xfce Desktop Environ-

ment and a regular X implementation. Version 9.2.148 of the CUDA toolkit is installed, and the proprietary NVIDIA driver (version 396.37) is used. The specific versions of all installed software can be found at the web page for this thesis [7].

Microarchitecture	CUDA Cores	L2 Cache	Base/Peak Clock	Peak FLOP/s (sp)
Kepler	960	384 KiB	980/1098 MHz	$1881.6 * 10^9$

Table 2.1: Information on the NVIDIA GeForce GTX660 GPUs used in BigEye.

Microarchitecture	Physical/Logical Cores	L3 Cache	Base/Peak Clock
Sandy Bridge	4/8	8 MiB	3.4/3.8 GHz.

Table 2.2: Information on the Intel Core i7-2600K CPU used in BigEye.

Part I

Mandelbrot Visualization

To familiarize ourselves with implementing interactive visualizations on the tiled display system, we conducted a preliminary study on visualizing the Mandelbrot set. This preliminary study allowed us to realize a visualization 'framework' enabling distributed computation and rendering using all of the tiled display system's GPUs. Since the Mandelbrot visualization problem can easily be solved in parallel, selecting it for the preliminary study allowed us to focus especially on the technical programming challenges related to this rather than on the complexities of parallelizing algorithms. The preliminary study is organized as follows.

In Chapter 3 we first introduce sequential and parallel algorithms to make drawings of the Mandelbrot set, as well as CPU and GPU implementations of these algorithms. We evaluate their performance in terms of running time, considering whether they enable real-time interactivity on the tiled display system. Next, in Chapter 4, we discuss the problems related to visualizing Mandelbrot drawings interactively on BigEye, the tiled display system used for this study. Based on this discussion, we propose a 'tiled visualization approach' in which each GPU in the system renders the visualization to be be displayed on the monitors directly attached to it. This approach is then evaluated for the Mandelbrot visualization application. The results of the preliminary study served as the starting point for our study of interactive network visualization on BigEye, which will be discussed in Part II of the thesis.

Chapter 3

Drawing the Mandelbrot Set

Before considering how to visualize the Mandelbrot set on the tiled display system, we discuss the Mandelbrot drawing problem more generally in this chapter. We present a sequential and a parallel Mandelbrot drawing algorithm, with implementations for both the CPU and the GPU, that we evaluate in terms of their running time for different image dimensions. For the parallel algorithm, we consider both a CPU implementation that uses SIMD instructions on all CPU cores, as well as an implementation on the GPU. The latter serves as the starting point for Mandelbrot visualization on the tiled display system, which will be discussed further in the next chapter.

3.1 Sequential Drawing Algorithm

The Mandelbrot set M consists of all complex numbers $c \in \mathbb{C}$, for which the sequence

$$|z_0|, |z_1|, |z_2|, \ldots$$

remains bounded, taking $z_0 = 0$ and $z_{n+1} = z_n^2 + c$. Since the Mandelbrot set is a subset of the complex numbers ($M \subseteq \mathbb{C}$), we can visualize it in the complex plane. To do so, we consider a discretized bounded area of the complex plane, which we refer to as the 'part' used for visualization. For each point c in this part, we then compute z_n for successive n, starting with $n = 0$. We stop increasing n once $|z_n| > 2$, which is a (well-known) criterion used to reduce the number of n values to evaluate, or if $n = n_{max}$, since $n \to \infty$ is infeasible. For all c that we evaluate in this way, we plot the final value of n, called the 'escape iteration', in the complex plane. Here the brightness of the point we plot corresponds to the value of n. In contrast to just depicting membership of M for each c in the part to be visualized, this reveals the geometrical structure of the Mandelbrot set in a more detailed way. Note that n_{max}, and the step-size used in discretization (δ) affect the trade-off between running time and approximation error. As $n_{max} \to \infty$ and $\delta \to 0$, the visualization starts to reflect the true Mandelbrot set, and the running time of the computation tends to infinity. Algorithm 2 presents the Mandelbrot drawing procedure described in this paragraph. The drawing width and height, w and h, correspond to the bounds of the part of the complex plane for which we visualize the Mandelbrot set, via the pixel size s, as follows: $w * s = \mathsf{Re}_{max} - \mathsf{Re}_{min}$, $y * s = \mathsf{Im}_{max} - \mathsf{Im}_{min}$.

Algorithm 2 Mandelbrot drawing

Input: w, h (drawing width and height, in pixels), s (pixel size in complex plane), C_o (origin of the drawing), n_{max} (maximum n to check).
Output: I ($w \times h$ rasterized drawing).

1: **for** $(x, y) \in \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$ **do**
2: $\quad c \leftarrow C_o - \frac{s*w}{2} + \frac{s*h}{2}i + x * s - (y * s)i$
3: $\quad n, z_n \leftarrow 0$
4: $\quad$ **while** $|z_n| \leq 2.0 \wedge n < n_{max}$ **do**
5: $\quad\quad n \leftarrow n + 1$
6: $\quad\quad z_n \leftarrow z_{n-1}^2 + c$
7: $\quad$ **end while**
8: $\quad I_{y,x} = \text{CMAP}(n)$ ▷ CMAP maps iteration to pixel color
9: **end for**

Figure 3.1 depicts the drawing resulting from Algorithm 2, with $C_o = -0.5 + 0i$ and $n_{max} = 255$. One of the characteristic properties of the Mandelbrot set is it its fractal geometry, i.e. self-similarity exists many different scales. Figure 3.2 demonstrates this, by showing that the overall structure can be recognized at in details of itself. For a more extensive discussion of the properties of the Mandelbrot set, we refer the reader to [12].

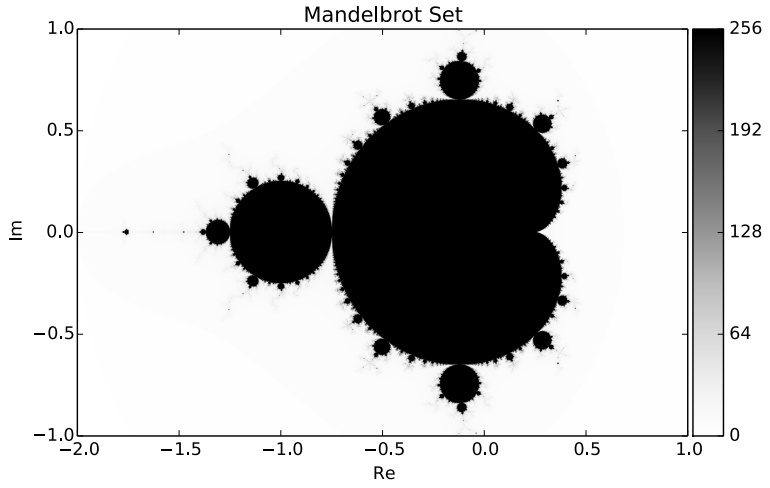


Figure 3.1: Drawing of the Mandelbrot set as computed using Algorithm 2, with $it_{max} = 255$. Escape iteration for each point corresponds to color via the index to the right of the drawing.

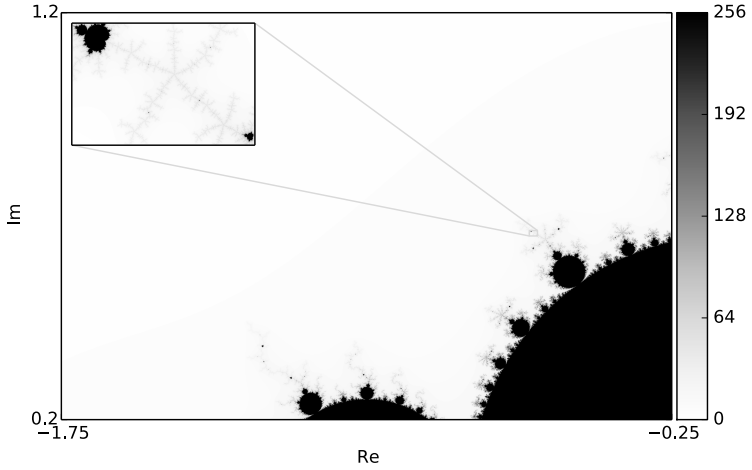


Figure 3.2: Self-similarity in the Mandelbrot set, the overall shape is present at multiple scales. Escape iteration for each point corresponds to color via the index to the right of the drawing.

3.2 Parallel Drawing Algorithm

The drawing algorithm presented in Section 3.1 can easily be parallelized by computing the values for pixels concurrently instead of serially. Consider Algorithm 3, which formalizes this by parallelizing the loop over (x, y) pairs.

Algorithm 3 Parallel Mandelbrot drawing

Input: w, h (drawing width and height, in pixels), s (pixel size in complex plane), C_o (origin of the drawing), n_{max} (maximum n to check).
Output: I ($w \times h$ rasterized drawing).

1: **parfor** $(x, y) \in \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$ **do**
2: $\quad c \leftarrow C_o - \frac{s*w}{2} + \frac{s*h}{2}i + x * s - (y * s)i$
3: $\quad n, z_n \leftarrow 0$
4: $\quad$ **while** $|z_n| \leq 2.0 \wedge n < n_{max}$ **do**
5: $\quad\quad n \leftarrow n + 1$
6: $\quad\quad z_n \leftarrow z_{n-1}^2 + c$
7: $\quad$ **end while**
8: $\quad I_{y,x} = \text{CMAP}(n)$ ▷ CMAP maps iteration to pixel color
9: **end parfor**

There exist no dependencies between computations for different pixels, therefore all (x, y) pairs can be evaluated concurrently, parallelizing the computation without requiring communication and synchronization. This is essential to achieve maximum performance on most parallel architectures, which are expected to provide significant performance benefits over serial proces-

sors. However, true peak performance might not be obtained on parallel architectures due to uneven load balancing between the parallel computations. This is because not all pixels require iteration up to the same value of n. Note that we assume that parallel computations can concurrently access the same memory to store results (cf. ln. 8 of Alg. 3), such that a reduction-phase which merges partial results from different memories to a single memory is not required.

3.3 Implementations of the Drawing Algorithms

This section discusses CPU and GPU implementations of the drawing algorithms that were presented in the previous section. We evaluate each implementation in terms of its running time, for increasing image dimensions. In doing so we consider if real-time visualization is possible at BigEye, the tiled display system used for this study. We deem an implementation suitable for real-time visualization if it is able to compute drawings at the refresh rate of the monitors used for visualization. Since the monitors in BigEye refresh their contents sixty times per second, this bounds the maximum running time of the desired implementation to $\frac{1}{60}\text{s} \approx 16.67\text{ms}$. This is an upper-bound since computed drawings also need to be transferred to the framebuffers of the monitors for display, and since the code might be interrupted, to run some other processes on the system. The time required to display drawings is discussed in Section 4.1, in the remainder of this chapter we focus solely on the time required to generate the drawings.

Since the running time of the drawing algorithm depends on the part of the complex plane for which we visualize the Mandelbrot set, as some pixels require further iteration than others, we we opt to measure worst-case running times. This corresponds to visualizing a part of the complex plane that requires iteration up to n_{max} for all pixels. To achieve true real-time performance, we need to consider this case since it is possible that the user navigates through such a region whilst exploring the Mandelbrot set. Thus, for all measurements, we set origin $O_c = 0$ and pixel size $s = w/0.1$, since this corresponds to the worst-case scenario. We set $n_{max} = 255$, since we found this to provide a good compromise between running time and approximation error.

All implementations we discuss are straightforward, and do not involve extensive optimization, since this is beyond the scope of this project. Besides the CPU SIMD code, which will be discussed later, all implementations are original and are accessible via the project web page of this thesis [7].

CPU Implementation

We first implement Algorithm 3 for the CPU using C++. Besides a serial implementation, we discuss a parallel implementation that uses *Single Instruction Multiple Data* (SIMD) instructions on all cores of the CPU. The 256-bit wide SIMD instructions we use, which are part of Intel's Advanced Vector Extensions (AVX) [29], can operate on up to eight 32-bit floating point numbers simultaneously. The SIMD code we use [53], assigns eight consecutive pixels, each from a different column, to a single SIMD register. Through SIMD instructions, these can then be operated on concurrently instead of sequentially. Assuming a linear speedup, vectorization by using the SIMD instructions can thus provide a performance improvement of 8× compared to a serial implementation. This might not be a realistic assumption, given potential side-effects that might result from using the SIMD instructions. Another degree of parallelism is introduced by using all CPU cores. Different rows of pixels are assigned to different CPU cores, by means of the OpenMP Application Programming Interface (API) [6]. Due to the 8 logical cores on the CPU in our system, this would again allow for a speedup of 8×, assuming a linear speedup is achieved. Parallelization through the use of SIMD instructions on all cores of the CPU thus results in a theoretical speedup of 64× over the serial implementation. Since this expects optimal hardware

utilization, which is not likely to be achieved using our straightforward implementation, we do not expect to achieve this speedup. Still, comparing the achieved speedup with the theoretical speedup gives an indication of the scalability of our parallel implementation.

We measured the average (wall-clock) running times of the implementations over 25 runs on BigEye, using an Intel Core i7-2600K CPU, with image dimensions ($w \times h$) ranging from 40×40 to 5120×5120 pixels. We consider the worst-case scenario, with $O_c = 0$, $s = w/0.1$, and we set $it_{max} = 255$. The CPU frequency governor was changed to the 'performance' setting for all cores, to stabilize the clock between measurements.

Dimensions ($w \times h$)	Serial (ms)	Vectorized (ms)	Vectorized + Multi-Core (ms)
40×40	1.65 (0.03)	0.19 (0.01)	1.07 (1.57)
80×80	6.60 (0.04)	0.73 (0.01)	0.13 (0.02)
160×160	26.32 (0.07)	2.91 (0.02)	0.51 (0.03)
320×320	104.40 (0.88)	11.49 (0.04)	2.51 (1.84)
640×640	417.24 (4.49)	45.73 (0.17)	9.10 (1.98)
1280×1280	1660.34 (6.02)	182.67 (0.19)	32.67 (2.04)
2560×2560	6629.43 (3.27)	733.24 (2.09)	129.38 (1.77)
5120×5120	26,521.85 (17.10)	2923.70 (4.55)	523.40 (5.97)

Table 3.1: Average running times for the CPU implementations of the Mandelbrot drawing algorithm, standard deviations between braces.

Dimensions ($w \times h$)	Vectorization Speedup	Vectorization + Multi-Core Speedup
40×40	8.8×	1.5×
80×80	9.0×	49.6×
160×160	9.0×	51.3×
320×320	9.1×	41.6×
640×640	9.1×	45.8×
1280×1280	9.1×	50.8×
2560×2560	9.0×	51.2×
5120×5120	9.1×	50.7×

Table 3.2: Speedups for the average running times in Table 3.1, in comparison to the serial implementation.

As the results in Tables 3.1 and 3.2 show, parallelization through SIMD instructions and multi-threading results in significant performance improvements. Combined they results in a speedup of approximately 51× over the serial implementation, as the image dimensions increase to 5120×5120 pixels. Interestingly, the speedups for the SIMD implementation exceed the linear speedup of 8×, for which we currently do not have an explanation. The speedup achieved through using SIMD instructions alone is approximately constant for all problem sizes, whereas the speedup for the multi-core SIMD implementation increases as the problem size increases. We explain this as a result of the overhead introduced by managing and scheduling multiple threads for the Multi-Core implementation. For small problem sizes this overhead likely constitutes a significant part of the running time whereas it becomes negligible as the problem size increases.

Using the CPU implementation of the Mandelbrot drawing algorithm, real-time performance at 60 fps becomes impossible as we increase image dimensions to 1280×1280 pixels. Since the tiled display system considered for this study has a resolution of $5,760 \times 4,320$ pixels, the CPU implementation clearly will not suffice.

GPU Implementation

To scale the Mandelbrot drawing algorithm to higher resolutions, we next considered the GPU as a platform. The throughput-oriented architecture of GPUs provides a significantly greater degree of parallelism than the general-purpose CPU, which is optimized more to achieve low latencies on a wide range of computational problems. Since parallelization already proved to be effective on the CPU, we thus expect further parallelization on the GPU to provide additional speedups. We implemented Algorithm 3 for the GPU using CUDA C, mapping the many parallel threads of computation provided by the GPU to different (x, y) pairs. As such, the GPU can concurrently compute many different pixel values. Threads are mapped to (x, y) pairs in such a way that simultaneous memory accesses by threads located on the same GPU core generally address consecutive ranges of aligned memory. This allows for multiple accesses to be coalesced into single transactions, which is important to utilize the GPU's available memory bandwidth. Accesses by different threads to the lookup table (LUT) translating escape iterations to colors are not necessarily coalesced.

We evaluated our implementation on BigEye, using a single NVIDIA GeForce GTX660 graphics card. As for the CPU implementation, we measure the average (wall-clock) running times over 25 runs, ranging image dimensions ($w \times h$) from 40×40 to 5120×5120 pixels. The worst-case scenario is considered, with $O_c = 0$, $s = w/0.1$, and we set $it_{max} = 255$. Table 3.3 presents our results, and Figure 3.3 combines these results with our results for the CPU implementation.

Dimensions ($w \times h$)	GPU Time (ms)	GPU/CPU Speedup
40×40	0.04 (0.00)	26.9×
80×80	0.06 (0.00)	2.1×
160×160	0.15 (0.00)	3.4×
320×320	0.51 (0.00)	4.9×
640×640	1.96 (0.00)	4.6×
1280×1280	7.69 (0.20)	4.3×
2560×2560	28.05 (0.00)	4.6×
5120×5120	112.12 (0.01)	4.7×

Table 3.3: Average running times for the GPU implementation, with standard deviations between braces. Speedup (GPU/CPU) is compared to the average CPU SIMD+MC implementation.

As the results show, the highly parallel architecture of the GPU indeed allows for further performance improvements. We observe that the speedup provided by the GPU approaches 4.7×, compared to the multi-core SIMD implementation on the CPU, as the problem size increases. The initial decline in the GPU/CPU speedup is explained as a result of the initial increase in performance by the multi-core CPU implementation, as discussed in the previous section. The running time of the GPU implementation drops below the threshold for real-time performance once we increase the resolution to 2560×2560 pixels, whereas this occurred at a resolution of 1280×1280 pixels for the CPU implementation. Although this is an improvement, it is still not sufficient for real-time performance on the tiled display system we consider for this study, since it has a resolution of 5760×4320 pixels.

However, if we consider using all GPUs in the tiled display system, and if we assign each GPU to compute the part of the complex plane to be displayed on the attached monitors, the dimensions of the image computed by each GPU reduces to 1920×4320 pixels. If we repeat the experiments for this resolution, we measure an average running time of 35.50 ms, which corresponds to an upper bound on the framerate of approximately 28 fps. Although this falls short of true real-time performance at 60 fps, it might still provide satisfactory results. Hence we further explore this multi-GPU approach in Chapter 4.

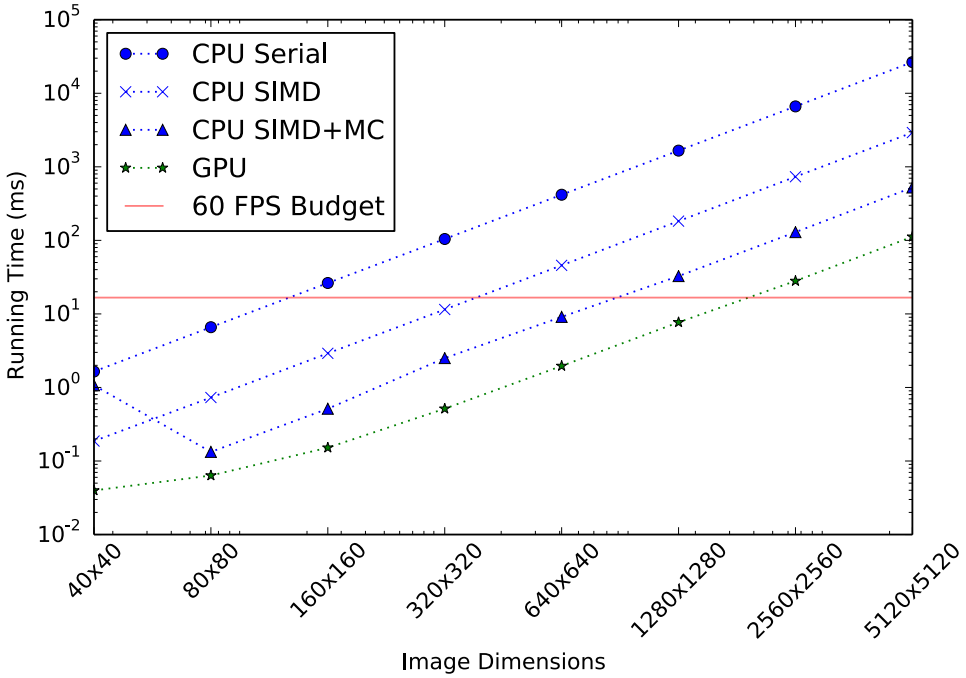


Figure 3.3: Average running times (worst-case) for the CPU (blue) and GPU (green) implementations of the Mandelbrot drawing algorithm. For the CPU a serial, vectorized (SIMD) and multi-core vectorized (SIMD+MC) implementation is evaluated.

3.4 Conclusion

In this chapter we evaluated a CPU and a GPU implementation of the Mandelbrot drawing algorithm. Each was evaluated in terms of its running time, to assess whether it is suitable for real-time interactive visualization on the 5760×4320 pixel tiled display system considered for this study. It was shown that parallelization allows the drawing algorithm to scale to significantly greater image dimensions. As image dimensions were increased to 5120×5120 pixels, parallelization on the CPU using SIMD instructions on all cores provided a speedup of approximately $51\times$ compared to a serial implementation. The GPU implementation provided an additional speedup of $4.7\times$ over this CPU implementation, corresponding to a $240\times$ speedup over the serial CPU implementation. Still, this was insufficient to allow a single GPU to generate drawings at 60 fps at the full resolution of the tiled display system. However, our results suggest that performance at 28 fps might be feasible if each GPU in the tiled display system is used to generate drawings for the monitors attached to it. We further investigate this multi-GPU approach in Chapter 4.

Chapter 4

Tiled Visualization Approach

Using the implementations of the Mandelbrot drawing algorithm derived in Chapter 3, we now consider how to visualize drawings of the Mandelbrot set interactively on BigEye, the tiled display system used for this study. In doing so, we aim for a system in which the user can explore the Mandelbrot set interactively through panning and zooming. Whilst this would be relatively simple on a typical desktop system, as we will discuss in Section 4.1, we face a number of challenges due to the tiled display system. First, as demonstrated in Chapter 3, generating visualizations at the full resolution of the tiled display system proves challenging. Also, updates to the different monitors have to be synchronized to ensure a coherent image across the display.

To solve these challenges, we propose and implement a 'tiled visualization approach' in which each GPU in the tiled display system renders the visualization to be displayed on the monitors directly attached to it. This also enables 'in situ' visualization, where the data computed by each GPU is visualized and displayed without transferring it to other parts of the system. The latter significantly reduces the amount of PCIe bandwidth utilization for each GPU. Finally, through a multi-threaded implementation of the approach, in which each thread addresses a different GPU, we are allowed to improve the coherency of the image that is displayed, by synchronizing the commands that initiate updates to monitors connected to different graphics cards.

Before detailing this approach, we study how to interactively visualize the Mandelbrot drawings on a single-monitor machine in Section 4.1. Next, in Section 4.2 we discuss why the multi-monitor support provided by the Linux operating system does not scale the single-monitor implementation to all of the monitors in BigEye. Finally, in Sections 4.3 through 4.5, we describe our tiled visualization approach, and discuss our results from implementing the Mandelbrot visualization using it.

4.1 Single-Monitor Scenario

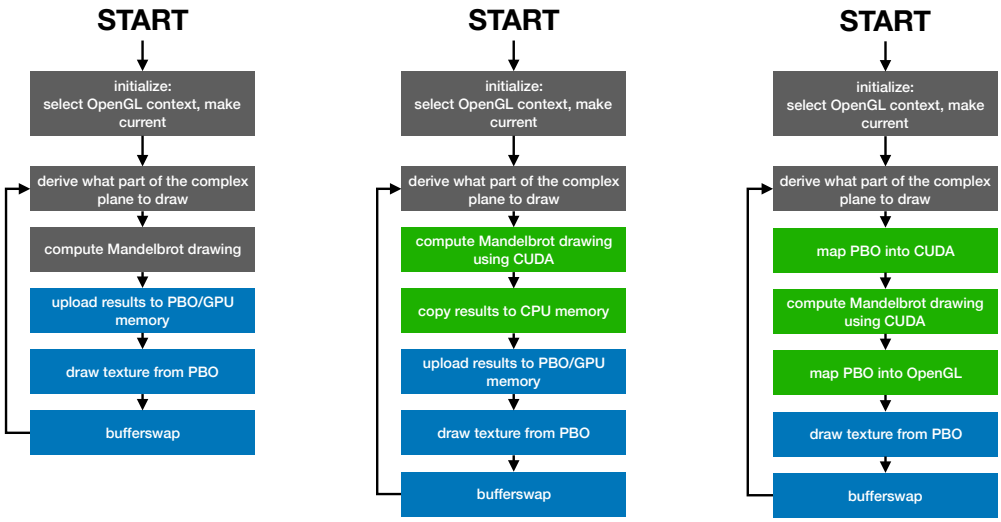
Before studying how to visualize drawings on the tiled display system, we consider a single-monitor desktop system. This allows us to isolate the time spent on displaying the Mandelbrot drawings, without considering any overhead resulting from using the multiple monitors and graphics cards in the tiled display system.

Display Loop and In Situ Visualization

To interactively visualize the Mandelbrot drawings, we use a display loop that continuously computes and displays a Mandelbrot drawing for the part of complex plane that the user wants

to view. The drawing is first computed using either the CPU or GPU implementation of the drawing algorithm, and stored in an OpenGL Pixel Buffer Object (PBO). The PBO is displayed on the monitor by setting it as the source of an OpenGL texture that is drawn to a window spanning the entire monitor. Since this window is a double-buffered drawable, all drawing occurs to an invisible back-buffer. The contents of this back-buffer are only displayed once its contents are exchanged with the visible front-buffer through a bufferswap. This bufferswap completes the display loop, which then proceeds with its next iteration by recomputing the Mandelbrot drawing.

For the GPU implementation, this design allows for 'in situ' visualization by mapping the PBO directly into CUDA memory. When computing the Mandelbrot drawing using CUDA, results can then be stored directly in the PBO, eliminating the overhead of transferring results from CUDA GPU memory, via CPU memory, to OpenGL GPU memory. Figure 4.1 depicts the display loop described in this paragraph, for both the CPU, GPU and in situ GPU approaches.



(a) CPU (b) GPU (c) GPU, in situ

Figure 4.1: Overview of the display loop for CPU, GPU and in situ implementations of the Mandelbrot drawing algorithm. Green boxes correspond to calls to the CUDA API, blue boxes correspond to calls to the OpenGL API.

To enable users to interact with the Mandelbrot drawing that is displayed, an 'event thread' waits for keyboard and mouse events to arrive. According to the keyboard and mouse interactions by the user the view-state, which represents the part of the complex plane for which the Mandelbrot drawing is computed, is updated. As such, a user can navigate around the Mandelbrot set through panning and zooming.

Display Time

We evaluate the time required to display computed Mandelbrot drawings for each variant of the display loop discussed in the previous section. This 'display time' includes the time spent updating the PBO, which is not needed for the in situ approach, up-to and including the bufferswap.

It thus excludes the time required to compute the Mandelbrot drawing. For our experiments, we temporarily disable synchronization between the bufferswap and the monitor's refresh, to exclude any time the bufferswap operation might spend waiting for the monitor's next refresh. This was achieved using the `EXT_swap_control` [48] extension to OpenGL.

We measured the average (wall-clock) display time using BigEye, which was configured to use only one of the 1920×1080 pixel monitors, over the first 25 iterations of the display loop. The dimensions of the Mandelbrot drawings were ranged from 40×40 to 5120×5120 pixels. To stabilize the CPU clock rate between different measurements, we set the CPU frequency governor for all CPU cores to the 'performance' setting. Figure 4.2 presents our results.

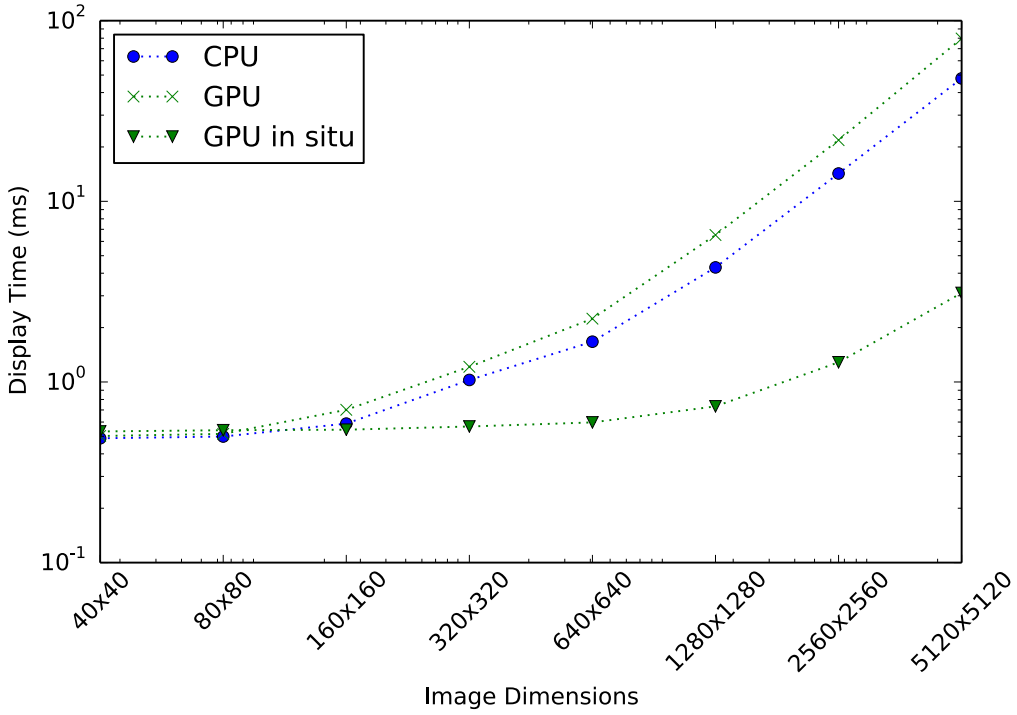


Figure 4.2: Average display time for the CPU, GPU and in situ GPU display loops.

As one would expect, displaying results through an in situ visualization approach requires the least amount of time. Still, the display time for the GPU implementation is smaller than expected. Since the GPU display loop requires each computed Mandelbrot drawing to be transferred twice between GPU and CPU memory for display, whereas this has to be done once for the CPU display loop, we would expect the GPU display time to be twice that of the CPU implementation. We have no explanation why our results do not reflect this, other than that the transfer from CUDA GPU memory to CPU memory might be implemented more efficiently than the transfer from CPU memory to OpenGL GPU memory.

Conclusion

For image dimensions up to 640×640 pixels, the overhead of displaying computed Mandelbrot drawings is limited to milliseconds for all implementations of the display loop. As resolutions increase beyond this threshold, an in situ approach is an order of magnitude faster than the CPU and GPU implementations. Combining this with our results from Chapter 3, which showed that the GPU implementation of the Mandelbrot drawing algorithm scales to the greatest image dimensions, we conclude that the GPU in situ visualization approach is best suited to interactively visualize the Mandelbrot set on BigEye. As such we focus on this implementation for the remainder of the exploratory study.

4.2 Multi-Monitor Graphics on Linux

In this section we discuss the multi-monitor support provided by the window system and graphics card drivers on Linux, to evaluate if these can be used to scale the single-monitor implementation of the Mandelbrot visualization presented in Section 4.1 to span all monitors in a tiled display system. We focus specifically on the system available to us, which consists of a single machine running Ubuntu 16.04.3 LTS, with 12 monitors with a resolution of 1920×1080 pixels connected to it through three NVIDIA GeForce GTX660 graphics cards. Full details on the system, named BigEye, can be found in Chapter 2. For a more general overview of programming tiled display systems for visualization, we refer the reader to [34].

Window System Support

The *X Window System* [49] adopted by most Linux distributions has traditionally provided multi-monitor support through the *Xinerama* extension. In the X Window System, physical display devices can be represented by *X Screens*, which Xinerama can unify in a single logical X screen. This allows applications to address all physical display devices connected to the system through a single X screen, instead of multiple X screens. To the best of our knowledge, using Xinerama thwarts manually implementing distributed rendering using OpenGL on all GPUs, since the NVIDIA implementation of OpenGL exposes GPUs for accelerated rendering through the X screen to which they are associated. Merging all X screens into a single logical X screen through Xinerama thus makes it impossible to manually address different GPUs.

Besides Xinerama, the *X Resize, Rotate and Reflect extension* (RandR) also provides multi-monitor support for systems using multiple graphics cards, since RandR version 1.4 [18]. Unfortunately this version of RandR is not fully supported by latest version of the proprietary NVIDIA driver (390.42) for the graphics cards in our system [42].

Driver Support

Multi-monitor support is also provided by various graphics card drivers. Here we focus on the most recent version (390.42) of the proprietary NVIDIA driver that is available for the graphics cards in our system. Using the NVIDIA driver up to four display devices connected to a single graphics card can be presented as a single uniform display device to the X window manager. In doing so, a single frame-buffer is allocated for all monitors connected to a graphics card. Still, when using OpenGL, updates to the monitors connected to a GPU only synchronize with the refresh of a single monitor [41, 40]. Hence tearing artifacts can still occur between display devices.

Similar functionality is also available for display devices connected through multiple graphics cards. Unfortunately this is limited to three display devices, without synchronization, when

using the consumer-grade GeForce graphics cards in our system [39]. These restrictions do not hold when using graphics cards sold for professional use, such as NVIDIA Quadro products. However, depending on the number of displays and graphics cards, dedicated hardware might still be necessary to synchronize updates to displays that connect to different graphics cards.

Evaluation

To scale the single monitor implementation presented in Section 4.1 to all monitors in BigEye, we configure the NVIDIA driver to present all monitors connected to a given GPU as a single display device to X. This results in three X screens, each representing a column of monitors, which we subsequently combine into a single logical X screen using Xinerama. As such, regular X applications can run across all monitors in the tiled display system.

Unfortunately this resulted in two issues when used to run the single-monitor Mandelbrot implementation on all monitors in BigEye. First, navigating around the Mandelbrot set resulted in significant tearing artifacts between monitors connected to different graphics cards, as depicted in Figure 4.3. Moreover, we achieved a framerate of approximately 6.5 frames per second (fps). A multi-GPU implementation of the Mandelbrot drawing algorithm might improve performance, however even a linear speedup would not improve the framerate beyond $3 * 6.5 = 19.5$ fps. Since this still falls short of real-time performance, we do not consider this option at this point.

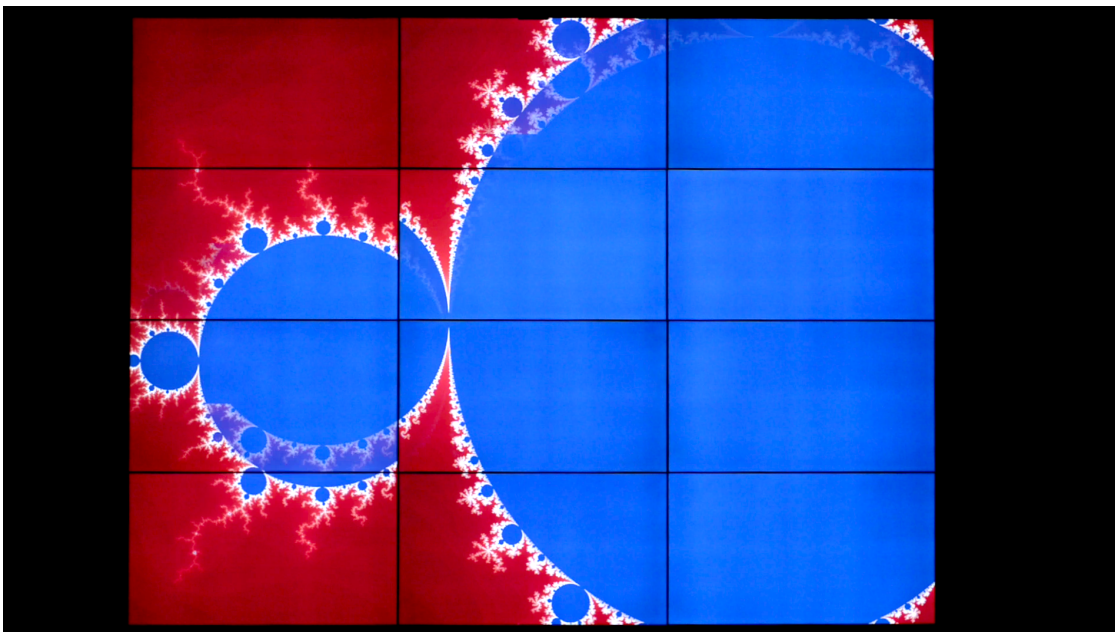


Figure 4.3: Tearing artifacts between different monitors.

4.3 Tiled Visualization Approach

To solve the performance and synchronization issues discussed in the previous sections, we propose to employ a 'tiled visualization approach' in which each GPU in the tiled display system renders the visualization to be displayed on the monitors directly attached to it. This allows

for 'in situ' visualization, where each GPU visualizes and displays the (GPGPU computed) data residing on it, without transferring this data to other parts of the system. For the Mandelbrot application we thus assign the part of the complex plane to be computed by a given GPU, to be the part that is displayed by that GPU. Benefits of the approach are that the computational power provided by all GPUs is employed to scale to the high resolution of the tiled display system. Also, no transfers over the PCIe bus are required to visualize results computed on the GPUs. Also, by individually addressing the different GPUs, we can potentially reduce tearing artifacts between different monitors.

To realize the aforementioned approach on BigEye, we configure the NVIDIA driver to provide a single framebuffer and X screen for monitors connected to the same graphics card. For BigEye, this results in three X screens, each representing a column of monitors. We display the Mandelbrot drawing for each of these columns using the 'in situ' display loop described in Section 4.1. This loop runs in parallel, on a different CPU thread for each of the graphics cards in the tiled display system. We synchronize the bufferswap calls between different threads through a thread-barrier, to improve the synchronicity of updates to different monitors in the system. Figure 4.4 illustrates this design.

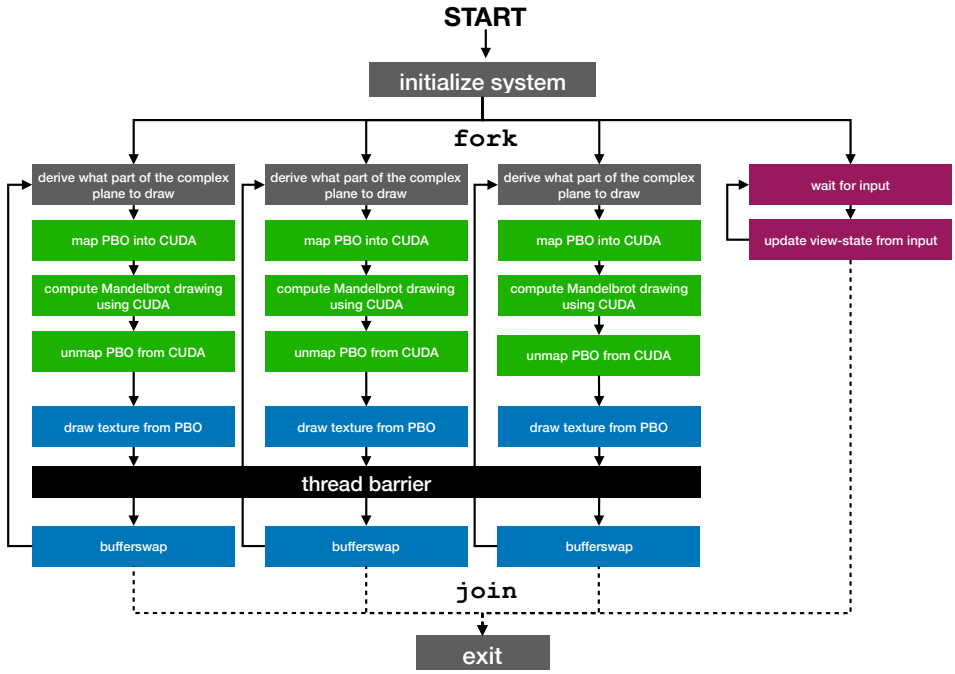


Figure 4.4: Overview of the tiled visualization approach applied to Mandelbrot visualization on BigEye, and implemented using multiple threads. Green boxes correspond to calls to the CUDA API, blue boxes to calls to the OpenGL API. Event-processing commands are depicted in purple. Branches indicated by dashed lines are taken when application should exit.

Note that each GPU operates on its own OpenGL and CUDA datastructures, without exchanging results with other GPUs or the CPU. Since this approach requires creating a distinct OpenGL context for each of the GPUs, we have to disable Xinerama.

In the remainder of this section we further discuss a number of implementation details regarding our implementation of the tiled visualization approach, concerning synchronizing updates to the different monitors in the system, issues related to input handling and multithreaded access to the X window system via a client library.

Monitor Synchronization

To ensure a coherent image on the tiled display system, updates to the monitors in the system need to synchronize to both their vertical refresh, as well as to updates to the other monitors in the system. If this is neglected, moving imagery can results in discontinuities, that appear as horizontal or vertical tearing artifacts between past and current input to the display. For an example of vertical tearing artifacts, see Figure 4.3. For our study we do not consider synchronizing updates to the monitors with their vertical refresh, but rather focus on synchronizing updates between the different monitors in the system. We deem the latter to be of greatest importance in ensuring a coherent image across the different monitors.

Intra-GPU synchronization, of updates to monitors connected to the same graphics card, is accounted for by the NVIDIA driver, given that we configure it to allocate a single framebuffer for all monitors connected to the same GPU, and given that we use double-buffered drawables. Since multiple graphics cards connect the monitors to the tiled-system, we also need to consider inter-GPU synchronization, i.e. synchronizing updates to monitors that connect to different GPUs. To this end we synchronize the function calls initiating updates for individual GPUs, between the different GPUs. That is, we synchronize the the OpenGL bufferswap across the different CPU threads executing the display loops. As shown in Figure 4.4, a thread-barrier blocks each thread from advancing until all threads are ready to perform the bufferswap. However, by default the NVIDIA driver synchronizes the OpenGL bufferswap operation for a given GPU to the vertical refresh of one of the monitors attached to it [40]. If the monitors selected for this across different GPUs are not synchronized in terms of their vertical refresh, this can introduce delays between the bufferswaps on different GPUs. These delays can take up to 16.67 milliseconds, assuming a refresh at rate of 60 Hz. Hence we disable the synchronization between bufferswap and vertical refresh via the `EXT_swap_control` [48] extension to OpenGL. The image corruption introduced by this is relatively minor and limited to one monitor per GPU. Besides, it potentially allows for significant improvements in the framerate that is achieved, given that the bufferswap operation no longer waits for the next refresh of one of the monitors. This is illustrated in Figure 4.5.

The inter-GPU synchronization approach presented in the preceding section leaves room for improvement, since it only synchronizes the initiation of the bufferswap processes for different graphics cards on the CPU, rather than the actual bufferswap operations, which occurs on the graphics cards. For our system, it appeared impossible to enforce a synchronized bufferswap at the level of the graphics card. The NVIDIA provided `NV_swap_group` [47] OpenGL extension that accounts for this, was not available on the consumer-grade GeForce graphics cards we use.

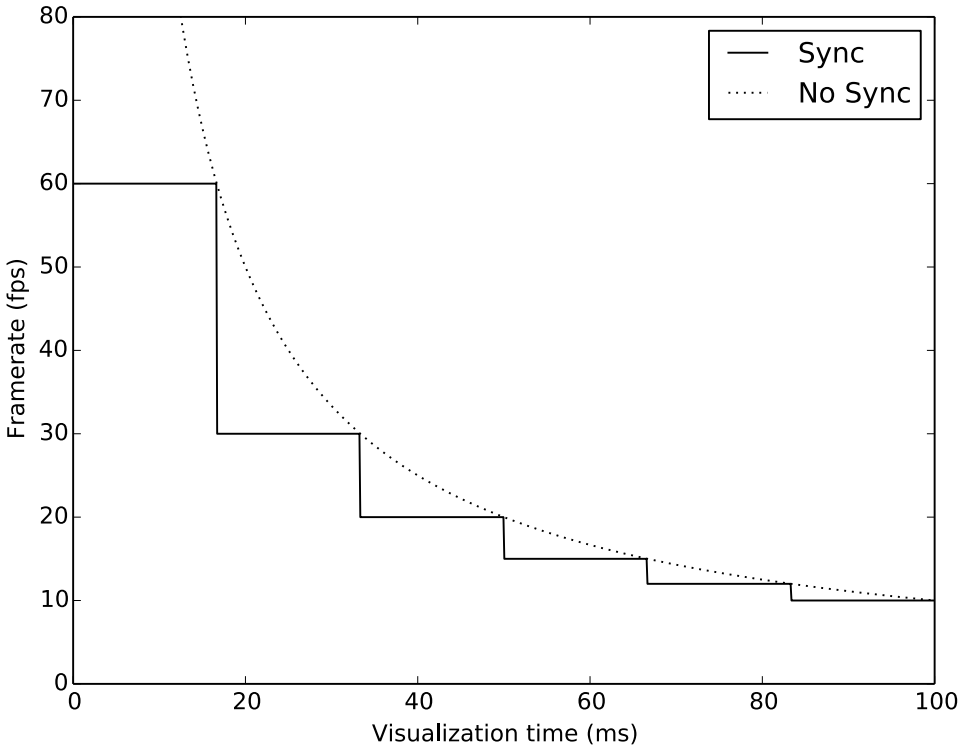


Figure 4.5: Relation between the time required to generate a frame (visualization time) and framerate, both with and without synchronizing bufferswaps with monitor refresh, for a monitor with refresh rate of 60Hz.

The Event-Loop

Threads executing the display loops, i.e. the display threads, and the thread executing the event loop, i.e. the event thread, concurrently access the view-state that represents the part of the complex plane that the user wants to view. As such, updates to the view-state from the event thread could be interleaved with reads by the display threads, during the generation of a single video frame. For example, imagine that for a given frame the display thread controlling GPU 2 first reads the view state. Next, the view-state is updated from the event thread, after which the display thread controlling GPU 1 read the view-state. If this occurs the different GPUs no longer agree on the part of the complex plane that should be visualized, each working on a different version of the view-state. This causes discontinuities in the visualization, as depicted in Figure 4.3.

We solve this by maintaining two copies of the view-state. The event thread only accesses one of these two copies, the 'event-view-state', which it continuously updates based on user input. The other copy, the 'display-view-state', is used by all display threads to determine what needs to be drawn to the display. A single display thread copies the event-view-state to the display-view-state, once all display threads have reached the thread-barrier. Display threads are released from the barrier only after this copy completes.

X client library

Since we rely on the X window system to access graphics- and input-devices, choosing the appropriate X client library proved crucial to successfully implement the proposed approach. It appeared that Xlib [17] was not suited to this. Although we explicitly enabled multi-threading support through the `XInitThreads` function, we observed that a call to `XNextEvent`, to retrieve events from the event-queue, blocked all display threads. This could not be resolved by using a distinct connection to the X server for each of the threads.

The 'X protocol C-language Binding' (XCB) [54] library did not present these issues. Using XCB, it was possible to address the X server from all threads through a single connection. As such, we used it for our implementation.

4.4 Results

We evaluate our implementation of the tiled visualization approach in comparison to the standard implementation that served as our starting point, i.e. the single-monitor in situ implementation presented in Section 4.1, running across all monitors of the BigEye system by means of Xinerama. Note that using Xinerama voids the in situ property, since one GPU generates the entire visualization, which is transferred over the PCIe bus for display on the monitors connected to other GPUs. In evaluating our results, we consider the two goals for the tiled visualization approach:

1. Improve visualization performance such that real-time interactivity, in which the displayed visualization reflects user input with the next monitor refresh, becomes a possibility. For the monitors in BigEye this corresponds to achieving a frame-rate of 60 frames per second (fps).

2. Achieve a coherent image across all monitors without tearing artifacts occurring as a result of moving imagery.

Regarding the first goal, we consider both the worst- and average-case frame rates achieved by both implementations. The worst-case performance is determined by viewing a part of the Mandelbrot set that requires iteration up to n_{max} for all pixels. We approximate the average-case frame rate by displaying the whole Mandelbrot set as it is depicted in Figure 3.1. Note that we disable synchronization between the OpenGL bufferswap and the vertical refresh of the monitors for both implementations. Table 4.1 provides our results.

	Baseline	Tiled Approach (Speedup)
Worst Case	4.4	25.2 (5.7×)
Average Case	6.5	47.9 (7.3×)

Table 4.1: Obtained framerates (fps) for Mandelbrot visualization on BigEye, using both the single-GPU Xinerama baseline and the three-GPU tiled approach.

As the results demonstrate, the tiled visualization approach improves performance significantly. Although true real-time performance at 60 fps is not achieved, we observe an average performance improvement of 6.5× between the two cases. The average case framerate of 48 fps enables more responsiveness than the framerate of 7 fps achieved using the baseline Xinerama implementation.

These performance improvements are in part due to using all GPUs in the system to compute the Mandelbrot drawings, without requiring inter-GPU communication over the PCIe bus to

display resulting drawings on all monitors. The latter is illustrated by Figure 4.6 which compares the PCIe utilization of each GPU for both implementations. We obtain this metric by querying the `GPUUtilization` attribute in `nvidia-settings` [44], which is provided by NVIDIA and which communicates with the proprietary NVIDIA driver.

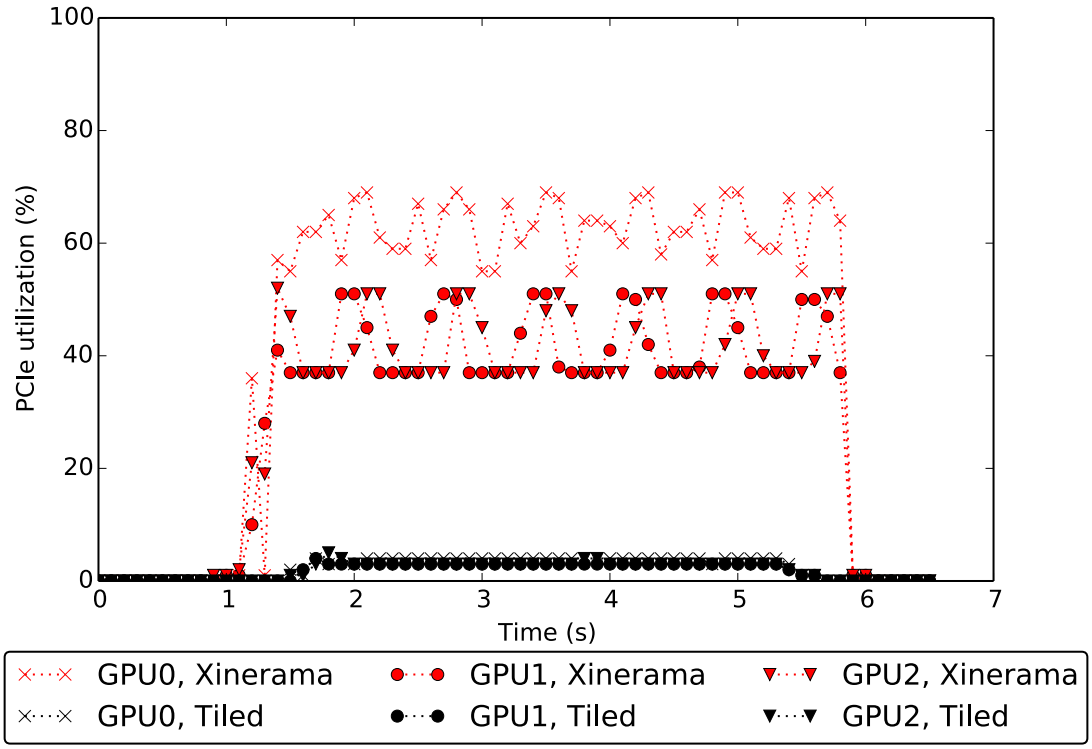


Figure 4.6: PCIe utilization for each GPU during two runs of the Mandelbrot visualization, once using Xinerama and once using the tiled visualization approach

The second goal, preventing tearing artifacts between different monitors, was approached by synchronizing the calls that initiate the OpenGL bufferswap for each of the GPUs. As discussed, this approach leaves room for improvement since it only synchronizes the initiation of the bufferswap process on the CPU rather than the actual bufferswap on the GPU. Still, our approach significantly reduces tearing artifacts compared to the implementation using Xinerama. When operating at high framerates, only slight tearing artifacts between different monitors could be observed. Tearing artifacts can be more significant when frame rates were reduced. The latter is an additional motivation to disable the synchronization of the OpenGL bufferswap and the vertical refresh of selected monitors.

4.5 Conclusion

In this chapter we presented and evaluated the tiled visualization approach in which each GPU in a tiled display system renders the visualizations to be displayed on the monitors directly attached to it. The approach was devised to improve over a standard solution using Xinerama in terms of performance, i.e. the framerate that can be achieved, and in order to reduce tearing artifacts between different monitors in the system. For our tiled display system, the approach enables interactive visualization of the Mandelbrot set at its full resolution of 5760×4320 pixels. Although true real-time performance at 60 fps was not realized, the worst-case framerate we achieved was 25.2 fps, which is a performance improvement of $5.7\times$ over an implementation which uses Xinerama to visualize Mandelbrot drawings generated using a single GPU. Besides improving performance, the tiled visualization approach also reduces the tearing-artifacts between different monitors to an acceptable level.

Part II

Network Visualization

The second part of this thesis presents our study of real-time interactive network visualization on BigEye, the tiled display system used for this study. We focus especially on large networks with tens of thousands of nodes and edges. Our starting point is the tiled visualization approach presented in the previous part of the thesis. Guided by this approach, an existing GPU implementation of the ForceAtlas2 graph layout algorithm has been adapted to run on multiple GPUs. This resulted in a system enabling real-time interactive visualization for networks with tens of thousands of nodes and edges, at a resolution of 5760×4320 pixels. To assess the interactive capabilities of the system, we implemented basic navigation and a number of interactions using the Nintendo WiiMote. This proved to be a more natural form of interaction on the wall-sized display than standard keyboard and mouse interaction.

The chapters in this part of the thesis are organized as follows. In Chapter 5 we first present the ForceAtlas2 algorithm, an existing GPU implementation of it, and a GPU accelerated network renderer that allows for the visualization of the layout process. Next, we evaluate the performance of the layout algorithm and network renderer to determine the components that might benefit from an implementation using multiple GPUs. The remainder of the chapter describes how we extend the ForceAtlas2 implementation and network renderer by using multiple GPUs, as well as the performance benefits this yields. In Chapter 6 we combine the distributed GPU implementation of ForceAtlas2 and the network renderer with the tiled visualization approach, to realize interactive network visualization on BigEye. Chapter 6 also presents our assessment of the interactive capabilities of the resulting system using the Nintendo Wii remote.

Chapter 5

Real-Time Network Visualization using multiple GPUs

We approach network visualization on BigEye by means of the tiled visualization approach described in the previous part of this thesis. To effectively use this approach, we thus require a network visualization code that runs on all GPUs in the system, which implements both the network layout algorithm as well as a network renderer that generates network drawings from the resulting layouts. In this chapter we describe our implementation of this, and we evaluate whether it meets the demands of real-time visualization for a range of real-world networks.

However, we first present the ForceAtlas2 graph layout algorithm that we will use for the remainder of this study, as well as an existing GPU implementation of it. Next, we describe how we implemented a network renderer using OpenGL, to visualize the layouts computed using the GPU implementation of ForceAtlas2. Together, the network renderer and ForceAtlas2 implementation comprise our network visualization code, which we evaluate in Section 5.2 on a range of real-world networks, to establish an initial threshold on the size of networks that can be visualized in real-time using a single GPU. In Section 5.3 we determine to what extend we can improve on this threshold, by adapting our network visualization code to run on multiple GPUs. The final two sections present a discussion of our results as well as our conclusions.

5.1 ForceAtlas2

For the remainder of our study, we focus on the ForceAtlas2 [30] graph layout algorithm, which we selected for a number of reasons. First, it is designed for interactive applications, due to its origins in the well-known Gephi [2] network analysis software. Second, we presented an open source implementation of the algorithm for the GPU in previous work [8], which will serve as the starting point for the visualization system presented in this study. Finally, the algorithm largely follows the general force-directed approach to graph layout, which potentially allows our results with ForceAtlas2 to extend to other force-directed graph layout algorithms.

Like other force-directed graph layout algorithms, ForceAtlas2 approaches the graph layout problem by considering the layout of a graph to be a physical system. Here nodes represent interacting physical bodies, and the layout process corresponds to a simulation of their interactions over time. An iterative procedure repeatedly displaces each node in the network in accordance to the resultant force acting on it, which is composed of repulsive forces away from all other nodes, attractive forces towards neighboring nodes, and a 'gravitational' force towards the origin

of the layout space. The repulsive forces between all node-pairs serve to move unrelated nodes away from each other, whereas the attractive forces between neighboring nodes should cause related nodes to move towards each other. The 'gravitational' force towards the origin of the layout ensures that disconnected components remain in proximity to the other components of the graph. To allow users to adapt the algorithm to their use-case, the magnitudes of repulsive and gravitational forces are parameterized. Note that this design does not aim to reflect any physical laws. Although this approach is similar to an n-body simulation, there are differences. For example, not the gravitational, but the the repulsive force in ForceAtlas2 corresponds to the gravitational force as it is used in n-body simulations. Yet, the gravitational forces in n-body simulations move bodies towards each other, whereas the repulsive forces in force-directed graph layout algorithms move nodes away from each other.

Since force-directed graph layout algorithms simulate the evolution of the graph's layout using discrete time-steps, a time-step size needs to be selected. ForceAtlas2 rephrases this problem as 'speed' selection, to make the concept more intuitive to users of the algorithm. High speeds correspond to long time-steps, and thus to a faster evolution of the layout, but with a lower precision. Low speeds correspond to small time-steps, and thus to a slower evolution of the layout, but with a higher precision. A given node's displacement, due to the resultant force acting on it, is proportional to the node's local speed. A node's local speed, in turn, is based on the extend to which it oscillates around a certain point, described as the node's 'swing', as well as a 'global-speed' affecting all nodes. The global-speed is adapted as the layout process advances in order for the layout to converge to some configuration. We do not discuss how the global speed is updated between consecutive iterations, as well as a number of settings that allow users to adapt the force-model employed by the algorithm, but refer the reader to [30] for more details.

A direct implementation of ForceAtlas2, as described in the preceding paragraph, would result in evaluating all node-pairs to calculate the repulsive force acting on each node. As such it would yield a computational complexity in $\Theta(|V|^2)$. Naturally this does not scale well to large networks. Repulsive forces between all node pairs are thus approximated using the Barnes-Hut algorithm [1], which employs a quadtree [1] representation of the graph layout to approximate the pair-wise force interactions. The quadtree is constructed by recursively splitting the layout space into four equally-sized rectangular cells, until no cell contains more than a single node. The resulting structure is stored as a quadtree with cells represented by the nodes in the quadtree, and leaves in the quadtree representing the nodes in the graph layout. See Figure 5.1 for an example. Each node in the quadtree holds aggregate information, such as the center of gravity, of the graph nodes nested in the sub-tree rooted at it.

To approximate the repulsive force acting on a given node in the layout using the quadtree, one traverses the tree from its root. The traversal of the quadtree stops at tree nodes whose center of gravity is too distant from the position of the graph node for which we traverse the tree. In this case the aggregate information on nested graph nodes is used. As such, the number of nodes to be considered when computing the repulsive force can be reduced from $\Theta(|V|)$ to $\Theta(\log|V|)$. Since the computational complexity of computing attractive forces is in $\Theta(|E|)$, and since gravitational forces can be applied in $\Theta(|V|)$ time, the computational complexity of one iteration of the ForceAtlas2 Algorithm is then in $\Theta(|V|\log(|V|)+|E|)$.

Algorithm 4 presents pseudocode for ForceAtlas2, which describes the procedure discussed in this paragraph more formally.

[1] For three-dimensional spaces an octree is used.

Algorithm 4 Pseudocode for a simplification of the ForceAtlas2 graph layout algorithm, adapted from [8]. Full details on the original ForceAtlas2 algorithm can be found in [30].

Input: Graph $G = (V, E)$, it_{max} (number of layout iterations), k_g (gravitational force scalar) and k_r (repulsive force scalar), θ (Barnes-Hut accuracy).
Output: For each $v \in V$, a position $\mathbf{p}_v \in \mathbb{R}^2$.

▷ Initialize variables
1: $global\_speed \leftarrow 1.0$
2: **for all** $v \in V$ **do**
3: $\quad \mathbf{p}_v \leftarrow \text{RANDOM}()$
4: $\quad \mathbf{f}_v \leftarrow (0.0, 0.0)^\top$ ▷ Net force on node v
5: $\quad \mathbf{f}'_v \leftarrow (0.0, 0.0)^\top$ ▷ $\mathbf{f}'_v$ is $\mathbf{f}_v$ of preceding iteration
6: **end for**

▷ Start layout process
7: **for** $i = 1 \rightarrow it_{max}$ **do**
8: $\quad \text{BH.BUILD}(\ )$ ▷ (Re)build Barnes-Hut tree
9: $\quad$ **for all** $v \in V$ **do**
10: $\quad\quad \mathbf{f}_v \leftarrow \mathbf{f}_v - k_g * (deg(v) + 1) * \mathbf{p}_v$ ▷ (Strong) Gravity
11: $\quad\quad \mathbf{f}_v \leftarrow \mathbf{f}_v + k_r * \text{BH.FORCE\_AT}(\mathbf{p}_v, deg(v), \theta)$ ▷ Repulsion
12: $\quad\quad$ **for all** $w \in \text{NEIGHBORS}(v)$ **do**
13: $\quad\quad\quad \mathbf{f}_v \leftarrow \mathbf{f}_v + (\mathbf{p}_w - \mathbf{p}_v)$ ▷ Attraction
14: $\quad\quad$ **end for**
15: $\quad$ **end for**
16: $\quad global\_speed \leftarrow \text{UPDATEGLOBALSPEED}()$
17: $\quad$ **for all** $v \in V$ **do**
18: $\quad\quad \mathbf{p}_v \leftarrow \text{LOCAL\_SPEED}(v) * \mathbf{f}_v$ ▷ Displacement
19: $\quad\quad \mathbf{f}'_v \leftarrow \mathbf{f}_v$
20: $\quad\quad \mathbf{f}_v \leftarrow (0.0, 0.0)^\top$
21: $\quad$ **end for**
22: **end for**

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

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

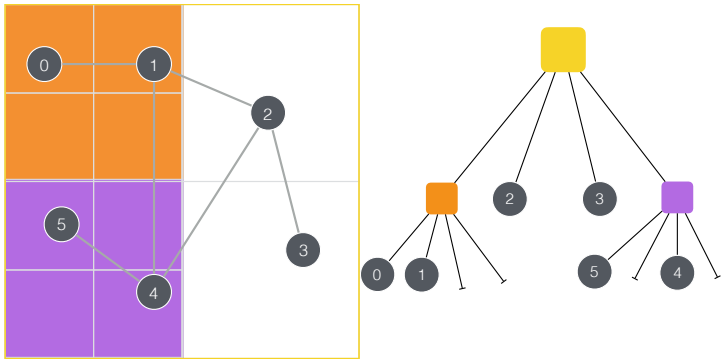


Figure 5.1: Network drawing (left) with corresponding quadtree (right), as it would be constructed for the purpose of repulsive force calculation using the Barnes-Hut[1] approximation algorithm.

5.2 GPU Implementation

Due to the parallel nature of force-directed graph layout algorithms, which involve many independent per-node force calculations, parallel implementations have the potential to scale to significantly larger networks than serial implementations. For our study we consider the open source CUDA implementation of ForceAtlas2 described in [8]. Figure 5.2 depicts the different CUDA kernels comprising this implementation, as well as how they relate to the pseudocode of the ForceAtlas2 algorithm presented in Algorithm 4. Note that the CUDA implementation of the Barnes-Hut algorithm used by this ForceAtlas2 implementation is due to [9]. For further details on these implementations, we refer readers to corresponding papers.

Drawing Networks Using OpenGL

To visualize the layout process, we extend the GPU implementation of ForceAtlas2 with a network renderer, implemented using OpenGL 4.1 (Core Profile) [50]. OpenGL was chosen since it is currently the only GPU accelerated graphics library whose data structures can be mapped into CUDA memory [46]. Since we use a CUDA implementation of ForceAtlas2, in situ visualization of the layout process can thus only be achieved using OpenGL. We represent the network data

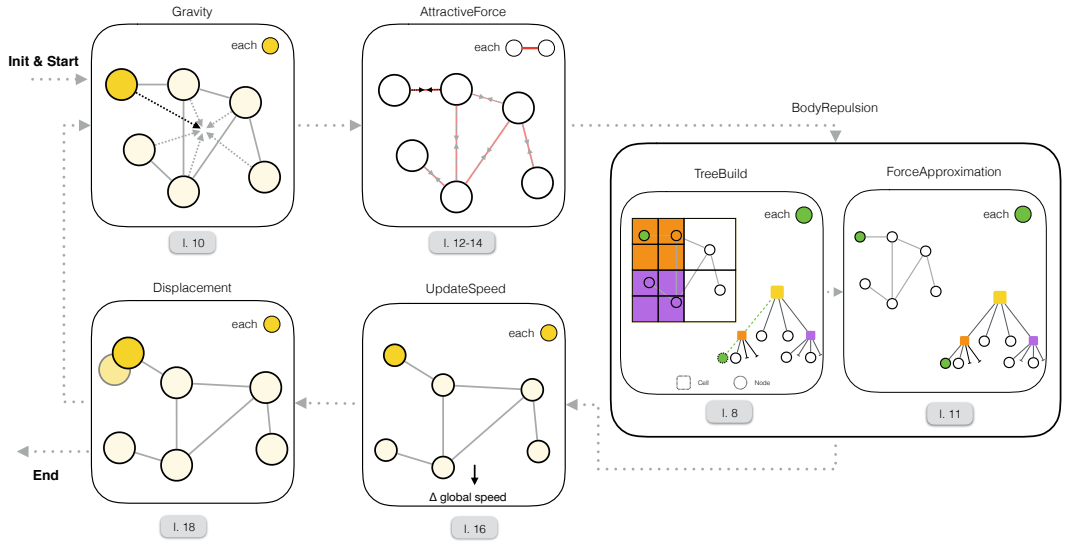


Figure 5.2: Different CUDA kernels (depicted as boxes) as they used in our GPU implementation of ForceAtlas2. For each kernel, the unit processed by CUDA threads is depicted (topright corner), as well as line numbers corresponding to the pseudo code in Algorithm 4 (gray boxes). Figure reproduced from [8].

in OpenGL as follows.

All node positions are stored in an OpenGL buffer object, which is indexed by node identifiers (node IDs) that range from 0 to $|V|$. Both the CUDA implementation of ForceAtlas2, as well as the OpenGL network renderer, operate directly on this OpenGL buffer. Additional node attributes can be represented in additional buffers, or can be interleaved with the data on nodes' positions. Such data could be accounted for in different stages of the OpenGL pipeline, and as such could be reflected in the appearance of nodes. For example, this would allow nodes to be sized or colored based on a property, such as their degree. The network's edges are also stored in an OpenGL buffer object, by consecutively storing the source- and target-node IDs for each edge. This data is uploaded only once to OpenGL, and is not shared with CUDA, given that it remains static throughout the layout process.

To draw the a network from this data, we use a shader-based approach. We draw the network's nodes by generating OpenGL point primitives, of which the positions are specified using the buffer object containing node positions. A fragment-shader is used to transform the square shapes that result from this into circles as described in [38]. The fragment shader implements this by reducing the opacity of each primitive's fragments based on their distance to center of the primitive by means of the `smoothstep` function. In contrast to simply discarding fragments beyond a certain radius, this ensures smooth rather than ragged circles. Edges are drawn as OpenGL line primitives, whose starting- and end-points are specified by using the buffer representing edges using their source- and target-IDs, as indices into the buffer storing node positions.

In order to enable user navigation, a vertex shader contains uniforms representing the current view. Using the vertex shader, all vertices are translated according to these uniforms, which are updated in correspondence to users their interactions.

Display Loop for Network Visualization

The network visualization approach described in the preceding sections, comprising graph layout using CUDA and network rendering using OpenGL, results in the the display loop depicted in Figure 5.3. As shown, networks are visualized by alternating between the CUDA implementation of ForceAtlas2 to advance the layout process, and drawing the results using the OpenGL network renderer.

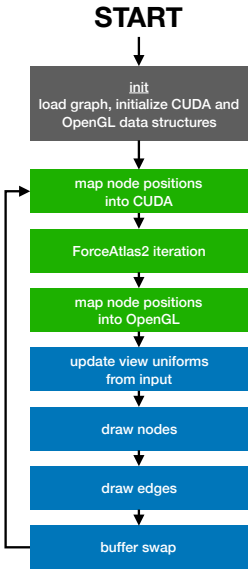


Figure 5.3: Display loop used for our network visualizer. Green boxes correspond to calls to the CUDA API, blue boxes correspond to calls to the OpenGL API.

Performance Measurements

We evaluated the network visualization approach described in the preceding section on a range of real-world networks, to establish a threshold on the size of networks that can be visualized interactively in real-time. As for the preliminary study on Mandelbrot visualization, we consider real-time visualization feasible when frames are generated and displayed at the refresh rate of the monitors. Since the monitors in BigEye operate with a refresh rate of 60 Hz., this bounds the time available time to generate a frame to $\frac{1}{60}$s $\approx$ 16.67ms. As such, the combined running time of the ForceAtlas2 layout algorithm and the network renderer must remain below 16.67ms. Note that, as for the Mandelbrot visualization problem, this is an upper bound since resulting drawings need to be transferred to the framebuffer for display, and since this time is shared with other processes on the system.

We evaluated our approach on BigEye using a single NVIDIA GTX660 graphics card. We use a range of real-world networks for our experiments, which are further detailed in Appendix A. We measured the average iteration duration over the first 500 iterations of the display loop,

setting $f_r = 80$, $f_g = 1$ and $\theta = 1$ with the gravity mode set to 'strong'. Drawings were rendered at a resolution of 5760×4320 pixels, matching the resolution of the BigEye tiled display system we use for this study. Note that we do not display drawings, but only consider the time required to generate them. Table 5.1 presents our results.

Network	Nodes	Edges	t_{fa2}	t_{dn}	t_{de}	t_{total}
CA-GrQc	4,158	13,422	1.79 (0.09)	0.13 (0.03)	1.92 (0.62)	3.85
petster	1,788	12,475	1.34 (0.05)	0.11 (0.03)	3.99 (0.42)	5.44
ppi_dip_swiss	3,766	11,922	1.82 (0.05)	0.13 (0.03)	3.67 (0.44)	5.62
PGPgiantcompo	10,680	24,316	3.84 (0.32)	0.21 (0.03)	2.90 (1.38)	6.95
ca-HepTh	8,638	24,806	3.68 (0.19)	0.19 (0.03)	4.44 (1.17)	8.31
dip	19,928	41,202	7.85 (0.36)	0.32 (0.04)	8.28 (2.14)	16.45
ca-CondMat	21,363	91,286	8.28 (0.39)	0.33 (0.04)	15.85 (4.44)	24.46
Newman-Cond_mat	22,015	58,578	9.64 (0.23)	0.34 (0.05)	16.14 (2.19)	26.12
ca-HepPh	11,204	117,619	4.64 (0.19)	0.21 (0.02)	26.27 (5.06)	31.13
wiki-Vote	7,066	100,735	3.33 (0.21)	0.17 (0.03)	32.77 (4.10)	36.26
ppi	37,333	135,618	14.16 (0.93)	0.52 (0.06)	22.35 (7.67)	37.03
GoogleNw	15,763	148,585	6.50 (0.35)	0.26 (0.03)	42.64 (10.11)	49.40
ca-AstroPh	17,903	196,972	7.22 (0.25)	0.29 (0.03)	43.51 (7.70)	51.02
email-Enron	33,696	180,811	12.56 (1.13)	0.46 (0.04)	44.89 (9.98)	57.91
p2p-Gnutella31	62,561	147,877	24.26 (1.12)	0.78 (0.08)	42.26 (5.57)	67.30
Brightkite	56,739	212,945	22.80 (1.22)	0.75 (0.07)	44.64 (9.65)	68.18
Cit-HepPh	34,401	420,784	14.26 (0.55)	0.49 (0.05)	63.14 (19.03)	77.88
Cit-HepTh	27,400	352,021	11.60 (0.40)	0.41 (0.05)	74.53 (16.44)	86.54
soc-Epinions1	75,877	405,738	30.90 (1.87)	0.93 (0.09)	123.24 (19.09)	155.08
email-EuAll	224,832	339,924	100.63 (25.76)	2.73 (0.11)	87.69 (37.80)	191.06
soc-Slashdot0902	82,168	504,230	35.62 (1.40)	1.02 (0.08)	182.22 (12.45)	218.86
Average			52.67%	1.43%	45.90%	

Table 5.1: Average running times for different components of the display loop used for network visualization. We report on the time spent on advancing the layout (t_{fa2}), drawing the nodes (t_{dn}) and the drawing edges (t_{de}). Times are in milliseconds, and averaged over the first 500 iterations. Standard deviations are denoted between braces and the dashed line indicates the real-time threshold.

The results in Table 5.1 demonstrate that the majority of the time in the display loop is spent on computing the layout (t_{fa2}) and on drawing the edges (t_{re}). The time required to draw the nodes (t_{rn}) is small, taking only a few percent of the total iteration time for most networks. We did not expect the edge drawing time to constitute a significant part of the total visualization time. Further analysis, see Figure 5.4, did not suggest a superlinear relationship between the number of edges and the edge drawing time. For future implementations of the network renderer we would consider the use of OpenGL triangle primitives to draw the edges, instead of using the provided line primitives. Given the prevalence of triangle drawing in most 3D computer graphics applications, to which the GPU platform and software is optimized, we hypothesize that this primitive might yield better performance than the line primitive.

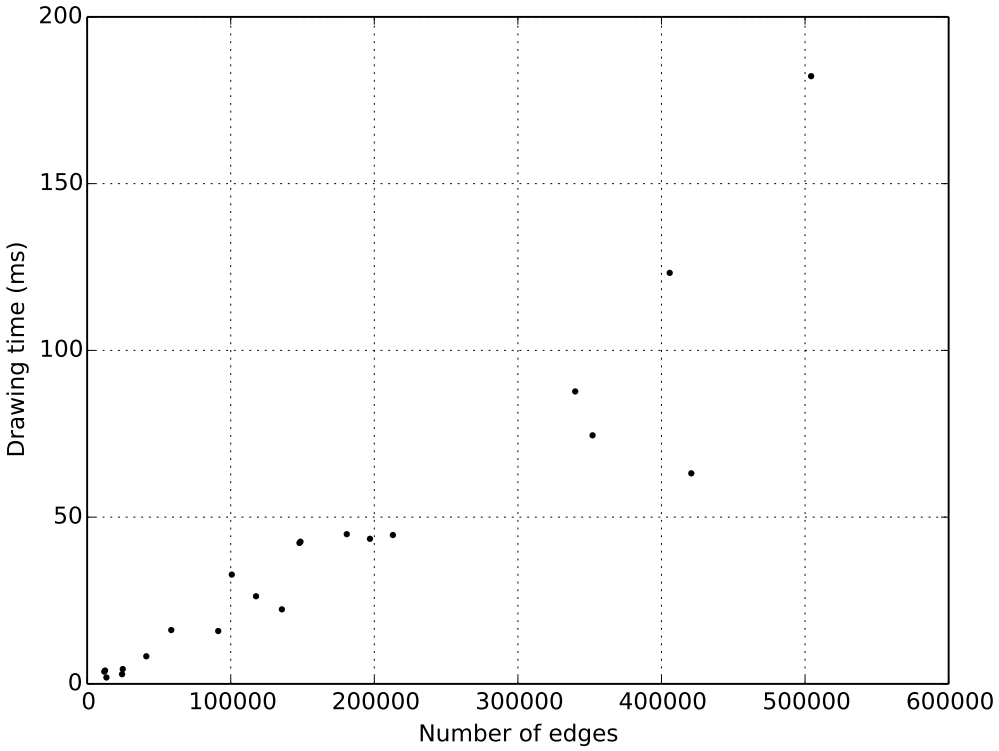


Figure 5.4: Relation between the time required to draw the edges in a network, and the number of edges.

5.3 Multi-GPU Implementation

To scale our network visualization approach to larger networks, we considered the performance benefits that could be obtained by using multiple GPUs. Given that the majority of the visualization time is spent on ForceAtlas2 and on network rendering, we focus exclusively on distributing these components between multiple GPUs. Our objective in doing so is to use the increased processing power provided by multiple GPUs, rather than using the increased amount memory that is available. This, given that the former limits the size of networks that can be visualized interactively in real-time. For example, the email-EUAll network, with 224,832 nodes and 339,924 edges, requires at most 100 MiB of memory, which is approximately 5% of the memory available on a single NVIDIA GTX660 graphics card. As such, we allocate a copy of all data structures on each GPU. Besides simplifying our implementation, duplicating data across different GPUs also reduces the need for communication between GPUs, potentially improving performance. Only the partial results computed by each GPU need to be transferred to the other GPUs.

We evaluate the degree to which the performance of our multi-GPU implementations scales with the number of GPUs using a dedicated multi-GPU machine of which we use six GPUs. Besides, we also evaluate the combined effect of our multi-GPU ForceAtlas2 implementation and network renderer on BigEye, using the using the same set of real-world networks used in

preceding experiments.

Distributing ForceAtlas2

As shown in Table 5.1, a significant part of the visualization time is spent on the ForceAtlas2 graph layout algorithm. To further analyze this result, we measured the running times for individual components of our ForceAtlas2 implementation using the NVIDIA `nvprof` profiler tool [45]. Our results for a number of datasets, presented in Figure 5.5, demonstrate that force approximation using the Barnes-Hut algorithm accounts for approximately 80% of the running time, on average. Hence, we initially focused on utilizing multiple GPUs to improve the performance of the force approximation component of ForceAtlas2.

To this end, we initially considered using the Bonsai [3] tree-code for force approximation, given that it is designed to operate on (large) distributed GPU systems. Although the Bonsai code is available as open source software, motivating our initial interest, incorporating it has proven to be too challenging and time consuming for the present study. This was mainly due to the time that was required to extract the subset of Bonsai's functionality that is required for graph layout. This involved, for example, adapting numerous data structures and procedures, each involving (very) low-level optimizations, to operate in a two-dimensional graph layout space instead of the three-dimensional space used in simulating the evolution of galaxies. This was not feasible within the time constraints for the present study. As such we decided to adapt the existing force approximation code [9] that is used in our ForceAtlas2 implementation, to run on multiple GPUs.

The repulsive force approximation computation was distributed between GPUs by partitioning the network's nodes into equally sized parts, and assigning a different part to each GPU. We partition the set of nodes after they have been sorted according to their spatial proximity, such that nodes assigned to the same streaming multiprocessor on a given GPU are in spatial proximity of each other. As described by the authors of the force approximation code [9], this is important to achieve good performance. After approximating the repulsive forces on each GPU, a reduction process takes place in which each GPU transfers its results to a designated master GPU. The master GPU aggregates the partial results, updates the global speed based on the results, and displaces nodes according to the force acting on them. Note that these three steps occur only on the master GPU. To complete the iteration of the ForceAtlas2 algorithm, the master GPU broadcasts updated node positions to other GPUs, such that each GPU again holds an up-to-date copy of all data. As shown in Figure 5.5 the speed-update and node displacement comprise a small fraction of the total iteration duration, which is why we do not consider distributing them between the GPUs.

Note that our work distribution does not assign the nodes displayed by a given GPU to that same GPU for repulsive force approximation. Although this would allow for true in situ visualization, it would pose a number of problems. For example, if nodes move between monitors connected to different GPUs, either as a result of user interaction or as the result of layout process, they would need to be assigned to a different GPU. Especially during the initial iterations of the layout algorithm, which involves large displacements for most nodes, this would result in significant amounts of inter-GPU communication. Besides, distributing nodes to different GPUs based on their location in the layout would not always allow for a balanced computational load between the different GPUs.

Scalability

We evaluate the scalability of our multi-GPU implementation on a dedicated multi-GPU machine. For our experiments we use six NVIDIA GeForce GTX 980Ti graphics cards, which are based on

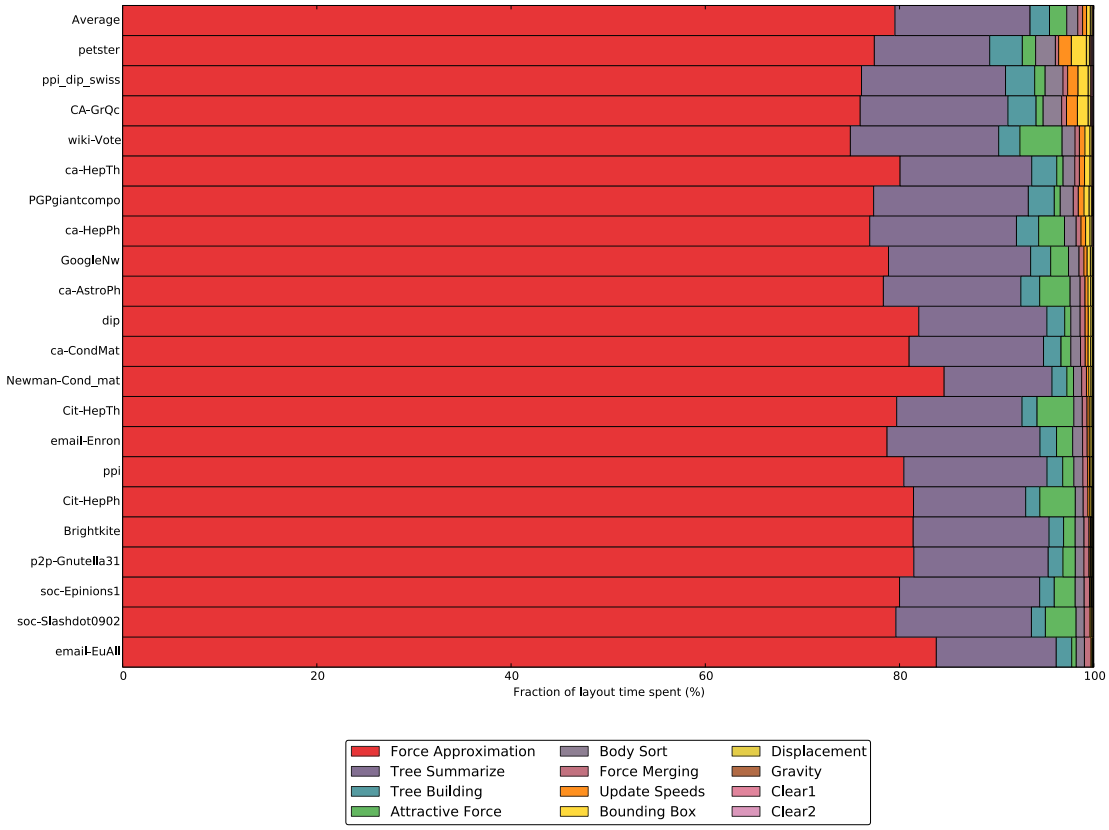


Figure 5.5: Fraction of the time spent in different components of ForceAtlas2, averaged over the first 500 iterations. Running times and numeric results can be found in Table B.1.

the NVIDIA Maxwell microarchitecture. The system runs CentOS 7.5, and uses the propriatary NVIDIA driver (version 396.26), and version 9.2.88 of the CUDA Toolbox. The peak performance of the GPUs in this system is significantly greater than the peak performance of the GPUs installed in BigEye, the tiled display system used for this study. As such we (temporarily) consider a set of larger networks for our experiments. As for the networks used in preceding experiments details on these networks can be found in Appendix A. In our experiments we measure the speedup of the average iteration duration, over the first 500 iterations of the layout algorithm, using up to six GPUs. We set $f_r = 80$, $f_g = 1$, $\theta = 1$ and set the gravity setting of ForceAtlas2 to 'strong'. Figure 5.6 depicts the speedups we observed, as well as the speedup that would from a linear speedup in the force approximation component that we distribute across multiple GPUs. The latter is derived from the average fraction of time spent on force approximation, across the different networks we evaluate, and denoted as 'Linear Speedup of 82.91%'.

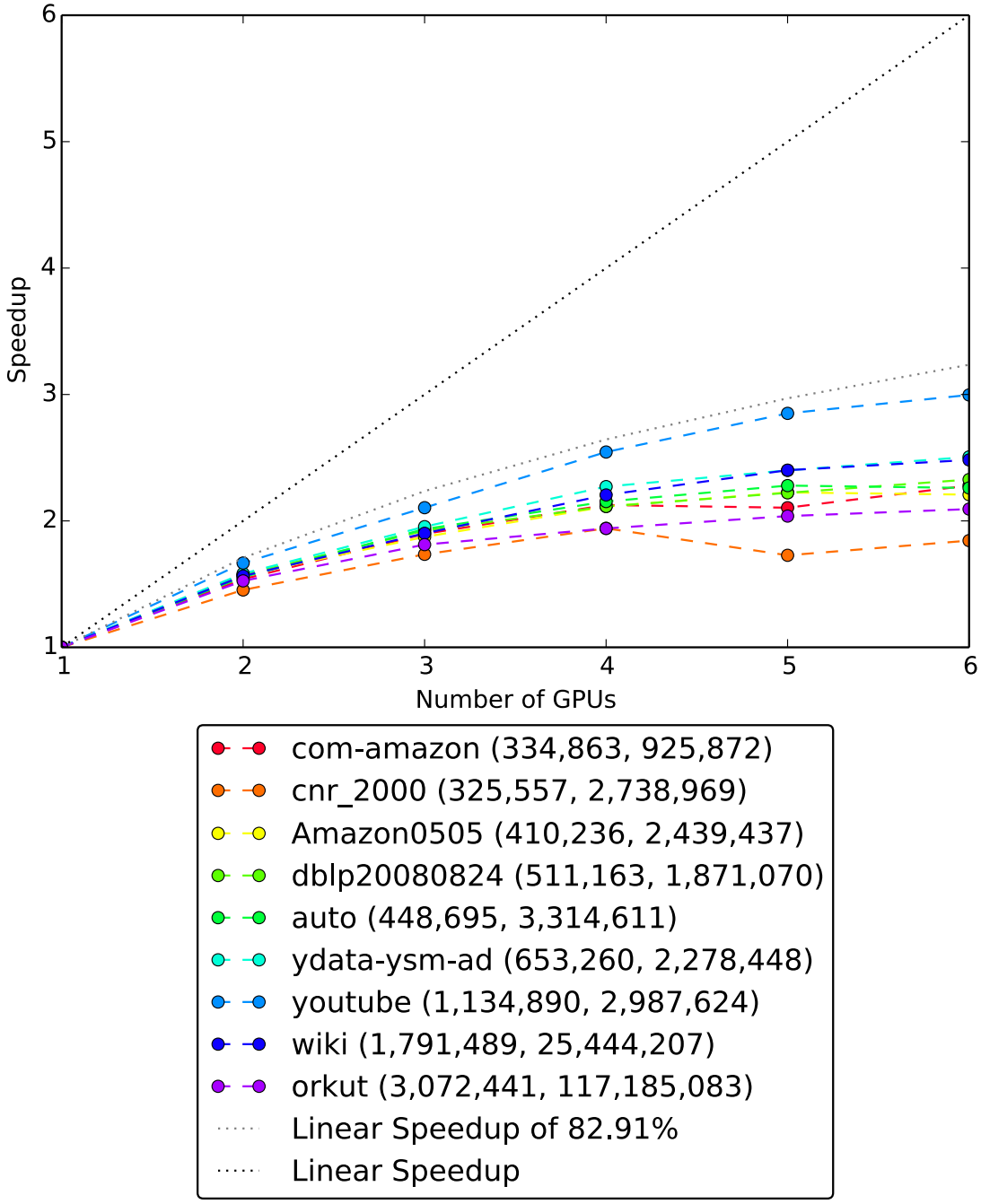


Figure 5.6: Performance scaling of our multi-GPU implementation of ForceAtlas2. Legend denotes network name and $(|V|, |E|)$.

As our results in Figure 5.6 depict, the speedup of ForceAtlas2 we obtained for the different networks generally follow the trend corresponding to the speedup that would follow from a linear

speedup of the force approximation component. The speedup gained by using additional GPUs diminishes as the number of GPUs increases, given that parts of ForceAtlas2 other than the multi-GPU force approximation start to dominate the layout time. Although this limits the scalability of our approach, this effect is less severe for our intended use-case with three GPUs. The obtained speedups generally increase with network size, which can be explained as the result of an increasing degree of parallelism being available to the system. Still, the 'wiki' and 'orkut' networks, the two largest we consider used for our experiments, yield small speedups given their size. We attribute this to the relatively large number of edges in these networks, i.e. their higher density, compared to the other networks. Indeed, as the single-GPU evaluation in Figure 5.7 shows, the wiki and orkut networks spend a larger proportion of the layout time on computing attractive forces between neighboring nodes. Given that only the repulsive force calculation is distributed between different GPUs, the speedup we obtain will be less for networks that spend more time in other components.

Figure 5.7: Fraction of the time spent in different components of ForceAtlas2 for the single GPU implementation, and the datasets and GPU model used in the scalability experiments. Derived from average running times for the different components over the first 500 iterations.

Multi-GPU Workload Composition

To determine how the multi-GPU implementation of ForceAtlas2 affects the fraction of time spent in different components of the algorithm, we reproduce Figure 5.5 for the multi-GPU implementation running on two of the GPUs in BigEye. The result, shown in Figure 5.8, illustrates that the the force approximation component still comprises the majority of the running time for a single iteration of ForceAtlas2. Still, as expected, the fraction of time spent in other components does increase. Figure 5.8 also shows the time spent on inter-GPU memory transfers is negligible compared to the time spent on the most time-consuming components.

We discuss the overall speedup obtained using the multi-GPU implementation of ForceAtlas2 on BigEye after discussing our distributed network rendering approach.

Distributing Network Rendering

Since rendering network drawings comprises approximately 46% of the visualization time, of which 45% is due to the time spent on drawing the network's edges, we also distributed the

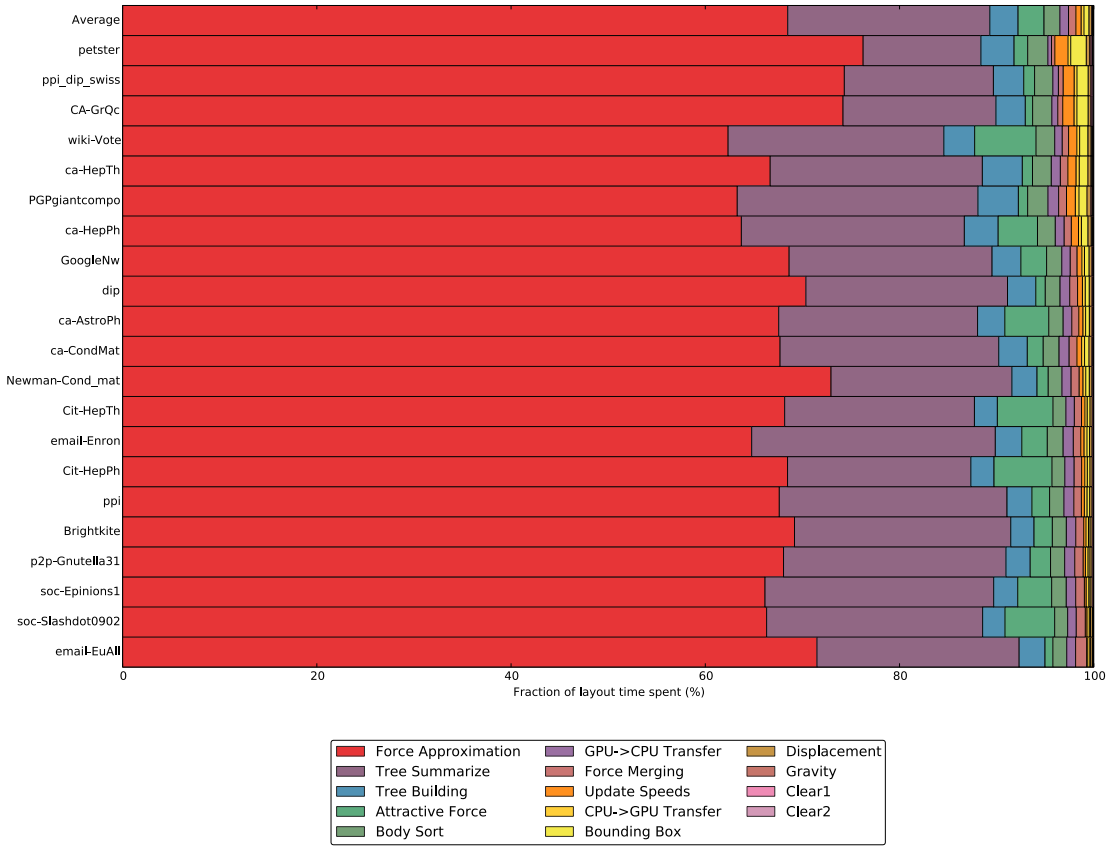


Figure 5.8: Fraction of the time a GPU spends in different components of ForceAtlas2, for the multi-GPU implementation running two GPUs in BigEye. Derived from average running times for the different components over the first 500 iterations.

network renderer across multiple GPUs. As for the distributed ForceAtlas2 implementation, all datastructures are duplicated on the different GPUs.

To distribute the render work across GPUs, the layout space was partitioned into equally sized columns, and each GPU was assigned a column for which to render the network drawing. See Figure 5.9 for an illustration. Although the same render commands are issued to each GPU, vertices on each GPU are translated differently to reflect the partitioning of the layout space. This causes each GPU to discard vertices which are part of the layout space assigned to other GPUs, via vertex clipping in the OpenGL pipeline.

We evaluated the scalability of our multi-GPU rendering approach using the same machine used to evaluate the multi-GPU implementation of ForceAtlas2. The same set of networks is used, and we again determine the speedup for the average iteration duration over the first 500 layout steps, setting $f_r = 80$, $f_a = 1$,, $\theta = 1$ and the gravity mode to 'strong'. During the layout process, we adapt the view after each iteration to ensure that the layout continues to span the image which we render. This is important to, for example, ensure that no parts of the layout get clipped and discarded, which would affect performance.

As our results in Figure 5.10 depict, performance scales up to six GPUs without significant saturation for most datasets. Notable are the reduced speedups when using uneven numbers of

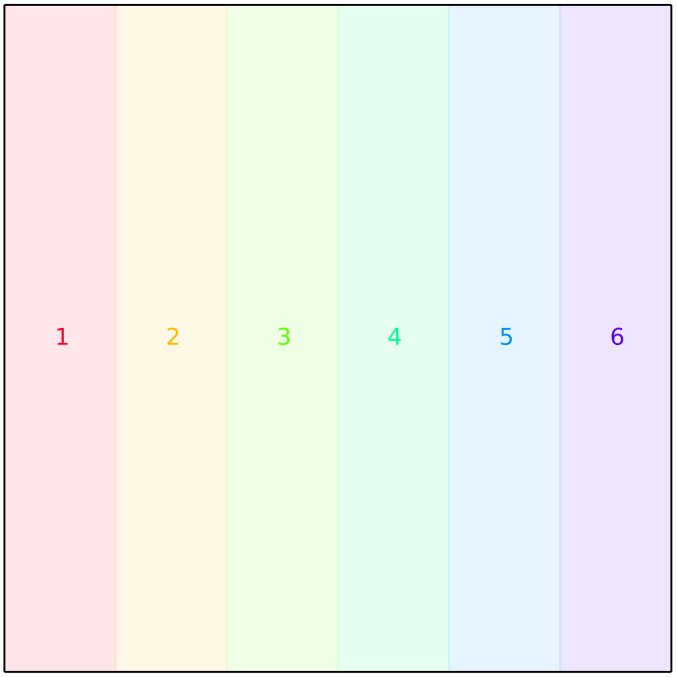


Figure 5.9: Partitioning of the layout when using six GPUs, numbers in parts correspond to GPUs responsible to render network in corresponding part.

GPUs. This effect is strongest when using three GPUs, and it diminishes as the number of GPUs increases. We explain this as the result of sub-optimal load balancing between the GPUs. When using an odd number of GPUs, the layout space is partitioned into an odd number of columns. As a result, the center of the layout will be assigned to a single GPU for rendering. Since the layouts that are generated are circular, there might be a tendency for most edges to cross the center of the layout, which would result in an increased amount of work being assigned to the GPU rendering the center of the layout.

Since the majority of the network rendering time consists of drawing the network's edges, see Table 5.1, we would expect networks with many edges to benefit especially from a multi-GPU implementation of the network renderer. This is not the pattern revealed by Figure 5.10. To investigate the cause of this, we considered the layouts for the 'com-amazon' and 'cnr_200' networks, since these are the networks that benefit the most, and the least from a multi-GPU implementation, respectively Figure 5.11 depicts the layouts for both networks. Clearly, the 'cnr_2000' network is more structured than the 'com-amazon' network. This might explain the differences in scalability for these networks. Although the layout for the 'com-amazon' network is not readable, it is uniform. This ensures a more uniform load-distribution between GPUs, given our work distribution.

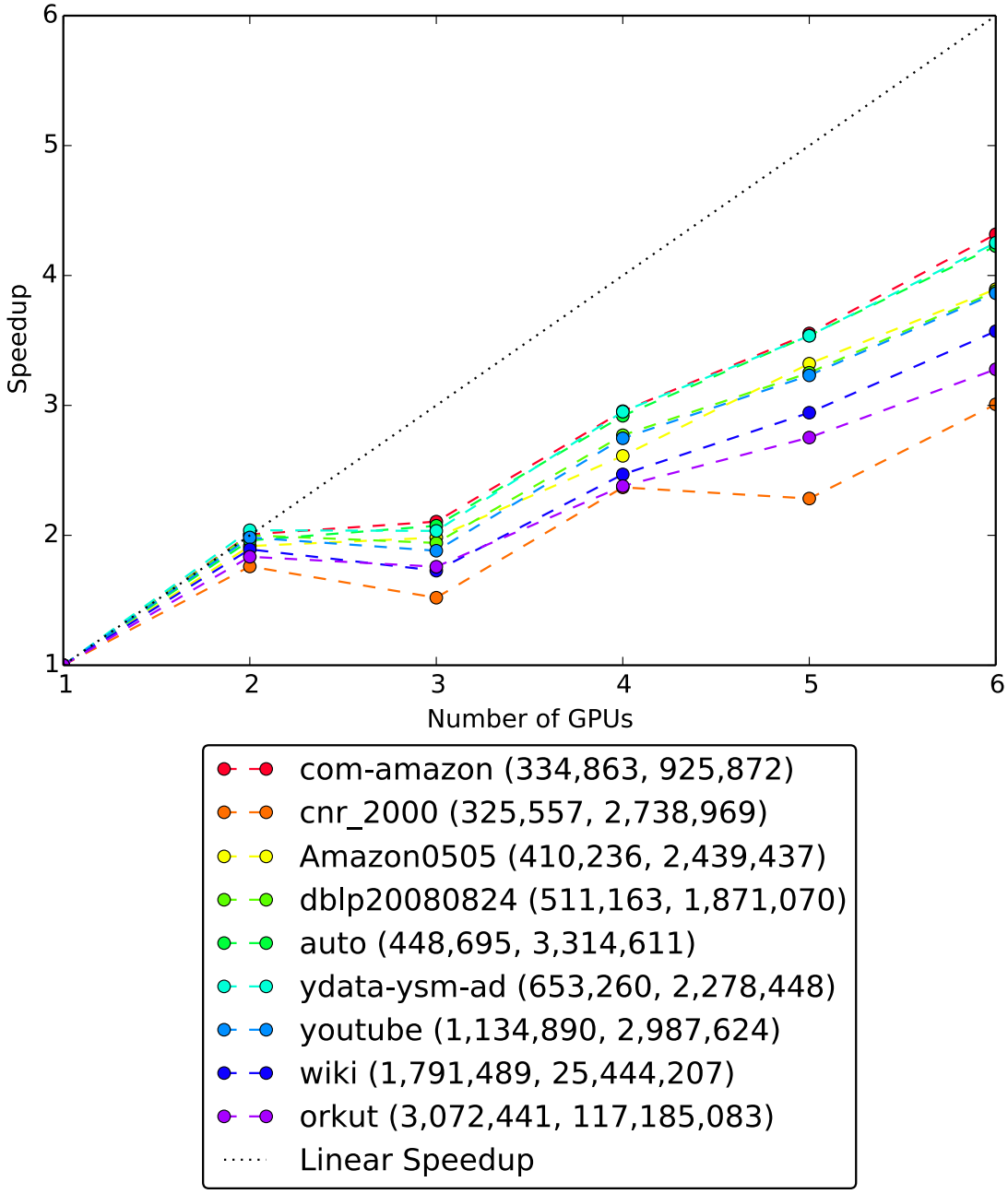
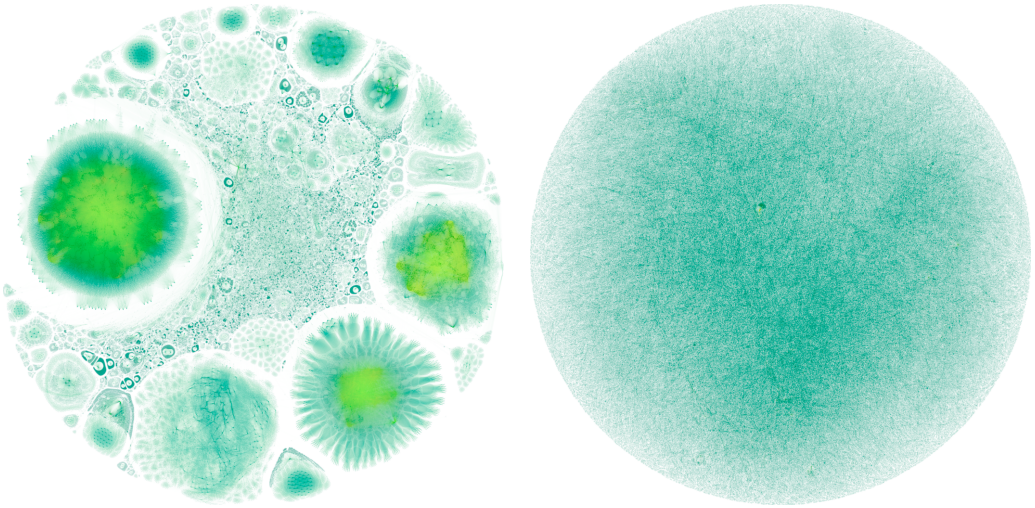


Figure 5.10: Performance scaling of our multi-GPU network renderer. Legend denotes network name and $(|V|, |E|)$.

Multi-GPU Visualization on BigEye

The combined performance improvements provided by the multi-GPU visualization approach on BigEye, involving both multi-GPU layout and rendering, are presented in Table 5.2. For our



(a) cnr_2000 (b) com-amazon

Figure 5.11: Layouts for two datasets from the experiments. Saturation of node colors correspond to degree.

experiments we used all of the three GPUs in BigEye and the same set of real-world networks used in our initial evaluation of the single GPU visualization approach, of which the results are presented in Table 5.1. Note that we now enable peer-to-peer data transfers between the GPUs in BigEye, thus bypassing CPU memory. This was not possible between all of the GPUs used for the scalability experiments, which is why we only consider this option at this point.

As shown in Table 5.2, the multi-GPU approach provides speedups for all networks we evaluated. As the network size increases, a speedup of approximately 1.7× is realized. Consequently, this extends the set of networks that can be visualized in real-time with the 'ca-CondMat', 'Newman-Cond_mat' and 'ca-HepPh' networks.

5.4 Discussion

We were able to obtain performance improvements for our network visualization approach through the use of multiple GPUs. However, we expect that significant improvements can still be made.

For future work on the distributed ForceAtlas2 implementation, we would initially consider improving the performance of the force approximation component. As Figure 5.8 reveals, the force approximation component of ForceAtlas2 continues to comprise the majority of the iteration duration, even after parallelization using multiple GPUs. Further performance improvements might also be realized by distributing additional ForceAtlas2 components, besides the force approximation computation, across multiple GPUs. As the scalability results in Figure 5.6 shows, this is crucial to ensure performance scales when using increasing numbers of GPUs. Distributing the computation of attractive forces between different GPUs might be considered first to account for dense networks. Finally, alternative force approximation approaches could be considered to improve performance. For example, the Bonsai code [3], which is designed to utilize multiple GPUs.

We suggest future work on the network rendering implementation to focus on two problems. First, the factors limiting the performance of the edge drawing implementation could be

Network	Nodes	Edges	t_{fa2} (Speedup)	t_{render} (Speedup)	t_{total} (Speedup)
CA-GrQc	4,158	13,422	2.00 (0.9×)	1.39 (1.5×)	3.39 (1.1×)
petster	1,788	12,475	1.62 (0.8×)	2.75 (1.5×)	4.37 (1.2×)
ppi_dip_swiss	3,766	11,922	2.00 (0.9×)	2.40 (1.6×)	4.39 (1.3×)
PGPgiantcompo	10,680	24,316	2.73 (1.4×)	1.95 (1.6×)	4.68 (1.5×)
ca-HepTh	8,638	24,806	2.63 (1.4×)	2.65 (1.7×)	5.28 (1.6×)
dip	19,928	41,202	4.99 (1.6×)	5.23 (1.6×)	10.21 (1.6×)
ca-CondMat	21,363	91,286	5.39 (1.5×)	9.33 (1.7×)	14.72 (1.7×)
Newman-Cond_mat	22,015	58,578	5.86 (1.6×)	8.87 (1.9×)	14.73 (1.8×)
ca-HepPh	11,204	117,619	3.45 (1.3×)	13.06 (2.0×)	16.51 (1.9×)
wiki-Vote	7,066	100,735	2.62 (1.3×)	18.91 (1.7×)	21.53 (1.7×)
ppi	37,333	135,618	7.85 (1.8×)	10.51 (2.2×)	18.36 (2.0×)
GoogleNw	15,763	148,585	3.96 (1.6×)	26.01 (1.6×)	29.97 (1.6×)
ca-AstroPh	17,903	196,972	4.57 (1.6×)	25.71 (1.7×)	30.28 (1.7×)
email-Enron	33,696	180,811	7.40 (1.7×)	26.88 (1.7×)	34.28 (1.7×)
p2p-Gnutella31	62,561	147,877	13.48 (1.8×)	24.50 (1.8×)	37.99 (1.8×)
Brightkite	56,739	212,945	12.42 (1.8×)	26.43 (1.7×)	38.84 (1.8×)
Cit-HepPh	34,401	420,784	7.69 (1.9×)	28.65 (2.2×)	36.33 (2.1×)
Cit-HepTh	27,400	352,021	6.96 (1.7×)	39.85 (1.9×)	46.80 (1.8×)
soc-Epinions1	75,877	405,738	17.30 (1.8×)	78.88 (1.6×)	96.19 (1.6×)
email-EuAll	224,832	339,924	51.27 (2.0×)	53.07 (1.7×)	104.34 (1.8×)
soc-Slashdot0902	82,168	504,230	20.05 (1.8×)	106.99 (1.7×)	127.03 (1.7×)

Table 5.2: Average layout (t_{fa2}), render (t_{render}) and total iteration times (t_{total}) obtained using our multi-GPU visualization approach with the three GPUs in BigEye. Times are in milliseconds, speedups are between braces and in comparison to the initial single GPU implementation. Real-time threshold indicated by dashed line.

assessed. We hypothesize that performance might be improved by using OpenGL triangle primitives instead of line primitives to draw the edges, given the prevalence of this procedure in most computer graphics applications. Second, the multi-GPU implementation could be improved by assuring a more uniform work distribution between the different GPUs. Given our results, we would consider partitioning the render work on a geometrical basis, e.g. by assigning individual edges and nodes to GPUs, rather than through the spatial partitioning we employed. The effect this would have on inter-GPU communication would need to be considered.

More thorough analysis of the relationship between network (layout) properties and the performance of our distributed network visualization approach could also be valuable in guiding the design of future systems.

5.5 Conclusion

In this chapter we evaluated our approach to real-time network visualization using a CUDA implementation of the ForceAtlas2 algorithm and a network renderer we implemented in OpenGL. We first evaluated a single GPU implementation of this approach. Our results showed that the majority of the visualization time is spent on repulsive force approximation in the ForceAtlas2 algorithm and on drawing the network's edges. As such these components were distributed between multiple GPUs. Our multi-GPU implementation is sub-optimal regarding scalability, and demonstrates the importance of uniform load balancing. However, it did show promising results for both the network renderer and force approximation algorithm. Using our approach, the network visualization time was reduced by approximately 1.7×, as network sizes increased, for the three-GPU system used for this study.

Chapter 6

Network Visualization on a Tiled Display System

In this chapter we describe how we achieved interactive network visualization on BigEye by combining the tiled visualization approach presented in Part I of this thesis, with the multi-GPU network visualization approach discussed in Chapter 5. The interactive capabilities of the resulting system are assessed by implementing a number of interactions using a Nintendo Wii remote. We evaluate our results in comparison to a standard single GPU visualization approach that uses Xinerama to display network drawings across all monitors in the tiled display system, and we suggests different approaches to further improve the visualization system.

6.1 Tiled Network Visualization

To realize interactive network visualization on BigEye, we combine the tiled visualization approach described in Chapter 4 with the multi-GPU network visualization approach discussed in Chapter 5. As shown in Figure 6.1, the architecture of the resulting system is very similar to the architecture of the Mandelbrot visualization system presented in Chapter 4. Three display loops, that run on different CPU threads, each address one of the GPUs in the system. For each GPU, the display loop repeatedly advances the layout algorithm, after which the part of the layout to be displayed on connected monitors is rendered and displayed. However, compared to the Mandelbrot implementation an additional synchronization point between the threads is required, besides the thread barrier that synchronizes the bufferswap.

Due to the work distribution used in the ForceAtlas2 implementation, force approximation results computed by the different GPUs have to be integrated before each GPU can proceed and render the network drawing to be displayed on the attached monitors. This integration step occurs on one of the GPUs, to which the partial results of other GPUs are transferred. The aggregated results are subsequently transferred to the other GPUs in the system. This is in contrast with the Mandelbrot application, where true 'in situ' visualization was employed, and each GPU rendered visualizations from data computed on the same GPU. Also, OpenGL is now used to render images, whereas for the Mandelbrot problem it only served to display the pixels that were computed using CUDA.

Figure 6.2 depicts our system in use on BigEye. We evaluate its performance in comparison to a baseline implementation, that uses a single GPU for the layout and rendering processes, and Xinerama to display resulting visualizations across all monitors of the tiled display system. The

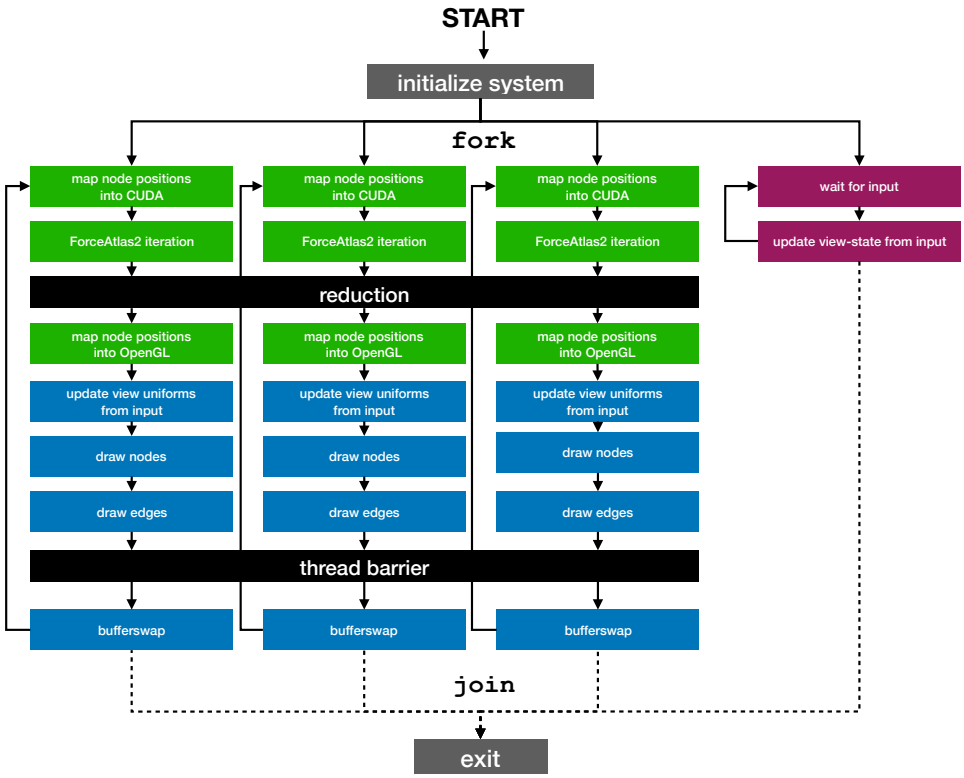


Figure 6.1: Network visualization on BigEye by means of the tiled visualization approach, implemented using multiple threads. Green boxes correspond to calls to the CUDA API, blue boxes to calls to the OpenGL API, and event-processing commands are depicted in purple. Branches indicated by dotted lines are taken when the application should exit.

baseline system uses the implementations of the layout algorithm and the network renderer that served as the starting point for the multi-GPU variants we discussed in Chapter 5.

We evaluate both systems on BigEye, comparing the average framerate over the first 500 frames. As for previous experiments, we set $f_r = 80, f_g = 1$, $\theta = 1$ and the gravity setting to 'strong'. After each layout step, we update the view such that the entire network drawing remains on the displays. Table 6.1 presents our results for a number of real-world networks.

As shown in Table 6.1, the performance improvements obtained using the tiled visualization framework, over a baseline implementation, approach 4.5× as the network size increases. This is in part due to the multi-GPU implementation of the layout algorithm and network renderer, as discussed in Chapter 5. However, we observe an additional speedup that can be attributed to the performance benefits of using the tiled visualization approach instead of Xinerama. Real-time visualization at 60 fps is now possible for an additional number of the networks, compared to the baseline implementation.

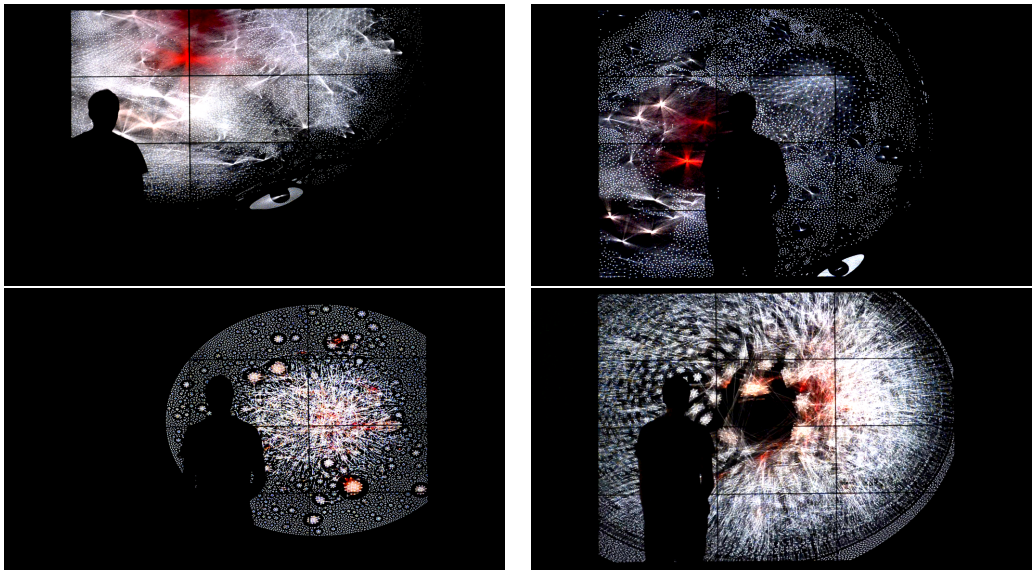


Figure 6.2: The resulting network visualization application running for two different networks on BigEye. Bottom-right image depicts the local repulsion mode.

Network	Nodes	Edges	Baseline	Tiled (Speedup)
CA-GrQc	4,158	13,422	82.04	228.96 (2.8×)
petster	1,788	12,475	64.93	200.97 (3.1×)
ppi_dip_swiss	3,766	11,922	64.91	182.68 (2.8×)
PGPgiantcompo	10,680	24,316	61.29	168.37 (2.7×)
ca-HepTh	8,638	24,806	51.10	155.86 (3.1×)
dip	19,928	41,202	28.07	88.28 (3.1×)
Newman-Cond_mat	22,015	58,578	15.86	62.34 (3.9×)
ca-CondMat	21,363	91,286	16.67	62.26 (3.7×)
ppi	37,333	135,618	11.30	49.85 (4.4×)
ca-HepPh	11,204	117,619	11.69	46.38 (4.0×)
wiki-Vote	7,066	100,735	9.38	45.81 (4.9×)
GoogleNw	15,763	148,585	7.05	33.10 (4.7×)
ca-AstroPh	17,903	196,972	6.86	31.85 (4.6×)
email-Enron	33,696	180,811	6.73	31.01 (4.6×)
p2p-Gnutella31	62,561	147,877	6.20	25.68 (4.1×)
Brightkite	56,739	212,945	5.90	24.37 (4.1×)
Cit-HepPh	34,401	420,784	4.17	23.96 (5.8×)
Cit-HepTh	27,400	352,021	4.03	21.14 (5.2×)
soc-Epinions1	75,877	405,738	2.39	10.74 (4.5×)
email-EuAll	224,832	339,924	2.58	9.64 (3.7×)
soc-Slashdot0902	82,168	504,230	1.68	7.13 (4.2×)

Table 6.1: Average framerates (in fps), over the first 500 frames, for the tiled visualization approach, in comparison to the baseline implementation using Xinerama.

6.2 Interaction

To assess the interactive capabilities of our network visualization system we implemented a number of interactions. All interactions were controllable by means of standard X keyboard and mouse events. Besides simplifying development, this provides an interface that many input devices can target. Basic navigation, i.e. panning and zooming, was implemented by means of mouse dragging and mouse scrolling, respectively, as well as via the keyboard's arrow-keys and the +/− keys. Next, we enabled control over a number of the network layout algorithm's parameters, as well as parameters for the network renderer, through the keyboard. As such users can adapt the scalars used for repulsive and gravitational forces during the layout process, and adjust the opacity of nodes and edges. The network renderer can also be set to color nodes based on their degree. Finally, we added three special interaction modes:

1. *Local repulse mode*, which causes the mouse pointer to act as a repulsive force. Nodes will move away from the pointer, with a displacement proportional to their distance to it and a scalar that the user can control through the keyboard. Using the local repulse mode, it is possible to destabilize the layout locally, potentially causing it to converge to a different configuration.

2. *Local heat mode*, which causes the local speed to increase for nodes near the mouse pointer. This increases the distance by which nodes around the pointer are displaced. As such, we expect it can also aid to further develop certain parts of the layout that may have converged to a sub-optimal layout.

For our evaluation of these interactions, we used a Nintendo Wii remote, see Figure 6.3, as input device. The WiiMote connects to the system via Bluetooth and reports on button presses, its acceleration along three dimensions and on its position in relation to infrared light sources that it tracks using an embedded camera. The latter allows the position of the WiiMote to be determined in relation to a 'sensor bar', which contains a number of infrared light sources. In our system the sensor bar is mounted underneath the middle column of displays.

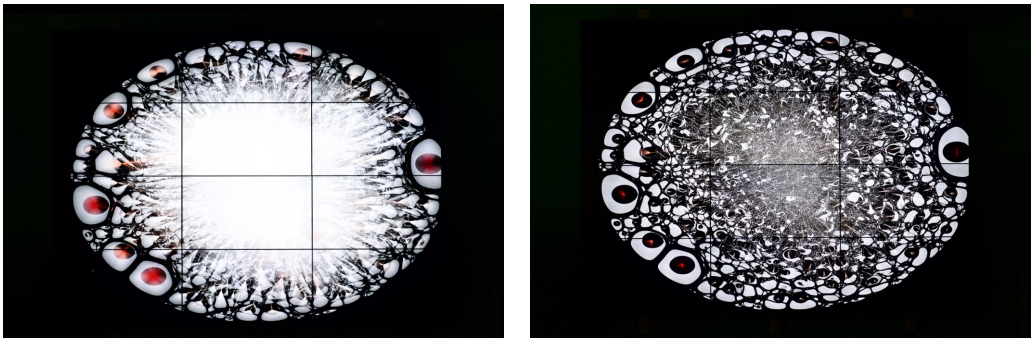
The open source XWiimote [25] software was used to convert WiiMote data into the keyboard and mouse events to be processed by our visualization system. We configured the X input driver provided by XWiimote such that the mouse pointer followed the location to which the WiiMote was pointed, and mapped the left mouse button to the button on the back of the WiiMote, which is controlled using the index finger. Buttons on the top of WiiMote were configured to control zooming, the opacity of nodes and edges, and the two interaction modes described in the preceding paragraph.

Since an evaluation of the human computer interaction (HCI) aspects related to these interactions is beyond the scope of this study, which focuses rather on the technical challenges related to facilitating interactivity, we do not consider any usability experiments. Still, for the reader's information, we will report on our personal observations, and provide a number of videos demonstrating our results on the web page associated with this thesis [7].

In our experience the WiiMote was effective at allowing users to navigate through networks. By means of simple point-and-drag gestures most users of the system were able to navigate around large networks, to reveal and inspect its substructures. We perceived that the wireless connectivity provided by the controller, enabling one to freely position oneself in front of the displays, was both effective and natural whilst exploring networks. Positioning oneself at distance of the displays improved the contrast between different parts of the network, causing the overall structure of the network to be better perceivable, most likely due to the blending of fine structures. The ability to interactively adjust the opacity of nodes and edges also proved to be

Figure 6.3: The Nintento Wii remote, also referred to as the WiiMote.

valuable in revealing the structure of the network. This is illustrated in Figure 6.4, which depicts how the structure of dense parts layouts can be revealed by reducing the opacity of edges.



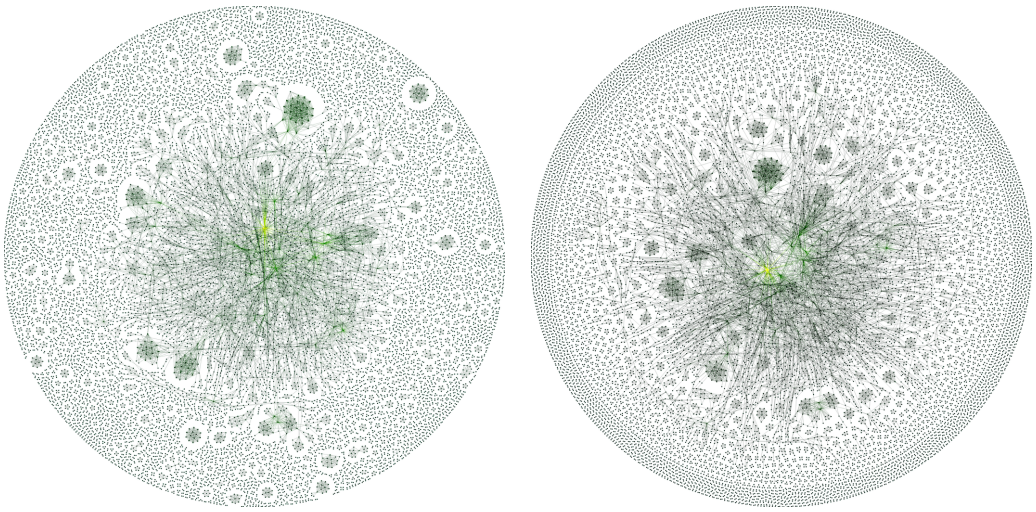
(a) Initial opacity for nodes and edges

(b) Decreased opacity for edges

Figure 6.4: Interactively adjusting the opacity of nodes and edges allow different structural properties of the network to be emphasized.

Regarding the two interaction modes, we found that the local repulsion mode indeed allowed users to force parts of the layout to converge to a different configuration. Based on the our specific experience it is not possible to conclude whether this enables better layouts in general. However, we did observe an interesting, and unintended, use for the local repulsion mode. As illustrated in Figure 6.5, the local repulsion mode can be used to arrange nodes based on their degree. When positioned at the origin of the layout, the local repulsion mode imposes a 'sorting' on the layout, forcing low-degree nodes to the periphery of the layout. This likely results from the combined effects of gravitational forces and the local repulsive force. Local repulsion forces nodes away from the origin of the layout, whereas the gravitational force moves nodes towards the origin, but with a magnitude proportional to the node's degree. The magnitude by which nodes are forced to the periphery of the layout is thus based on their degree. If a network consists

of different components, the components with a similar topology, are likely to be positioned in proximity of each other.



(a) Without local repulse (b) With local repulse

Figure 6.5: The same network visualized with and without a local repulsive force at the origin of the layout. Saturation of node color corresponds to degree.

The local heat mode proved to be less intuitive to use. Correctly configuring the temperature offset to be used near the mouse was important to prevent chaotic displacements of all nodes. Once a correct setting was found, nodes near the pointer were indeed displaced by a larger distance than other nodes in the network. However, in our experience this had no clear effect on the layout quality in this part of the network.

6.3 Discussion

The tiled visualization approach enabled performance levels suitable to real-time interaction for a number of networks of the network we evaluated. We successfully exploited this possibility in our assessment using the WiiMote. We consider our results a proof-of-concept, which is not directly applicable for data-exploration and -analysis. This is mainly due to the limited number of interactions we implemented. In the context of the standard *'overview first, zoom and filter, then details on demand'* information seeking mantra [51], we now provide *overview* for larger networks, but still lack *filter* and *details on demand*.

In future work our primary goal would thus be to realize interactions that allow user to obtain information on selected nodes and edges. This information could be presented in the form of labels, but we could also imagine that a dedicated 'inspector window' could be added, possibly on a separate desktop system, to present such information. The possibility to highlight the nodes and edges connected to a given node would be a good means to improve the system's capability to depict the topology of the network. The same holds for the possibility to dynamically change the appearance, e.g. color or size, of nodes based on different network properties. In order to enhance users' sense of context, a map of the layout indicating the part that is in view could be superimposed in one of the display's corners.

6.4 Conclusion

We described how the tiled visualization approach presented in Part I of this thesis was combined with the multi-GPU network visualization approach presented in Chapter 5, to achieve interactive network visualization on tiled display systems. Our experiments with the resulting system on BigEye, a 25 megapixel tiled display system equipped with three graphics cards, demonstrated a speedup approaching 4.5× as network sizes increases, over the single GPU baseline implementation we considered. Consequently, we achieved performance levels suitable to real-time interactive visualization of networks with tens of thousands of nodes and edges. Using the Nintendo Wii remote as input device, we assessed the interactive capabilities of the system. The result was an effective proof-of-concept system, that enabled real-time interactive exploration of large networks.

Chapter 7

Discussion and Future Research

Before discussing directions for future research, we consider our results in relation to the previous work on interactive network visualization using tiled display systems that was discussed in Chapter 2. We specifically consider the work by Mueller et. al. [37], given that it also describes a general-purpose interactive network visualization system.

Mueller et. al., consider an eight node cluster-based tiled display system composed of eight monitors, whereas we consider a single-node system with three GPUs connected to twelve monitors for the present study. Although Mueller et. al. do not report the resolution of their system, the photos in the paper do not suggest it exceeds the 25 megapixel resolution of the system considered for the present study. Mueller et. al. report frame rates of up to 5 frames per second (fps) for randomly generated graphs with 8000 nodes and (approximately) 80,000 edges. For a real-world graph of similar size we achieve a frame rate of approximately 60 fps. This corresponds to a performance improvement of 12×, at at least the same (but most likely higher) resolution.

Although we consider the results presented in this thesis a valuable addition to existing research, we expect improvements can be made regarding: (multi-GPU) performance, interactivity and synchronization between the monitors. In the remainder of this section we describe how we expect these improvements can be obtained.

- Performance improvements of the multi-GPU network visualization approach depend on improvements to the layout algorithm and renderer.
  - We expect performance improvements of the layout algorithm to be realized through improvements to the force approximation component and by distributing additional components between multiple GPUs. Force approximation comprises the majority of the time spent on computing layouts, even after distributing this computation over two GPUs. As such, it can be considered a performance bottleneck. In future research, we would evaluate the performance of the implementation we use for this component [9] against alternatives such as Bonsai [3] and a recent approach based on graph topology [36], to evaluate whether replacing it can be a means to improve performance.

    To improve the performance of our multi-GPU ForceAtlas2 implementation, we would also consider distributing additional components of the algorithm between the GPUs. For our current implementation we opted to only distribute the repulsive force approximation component between different GPUs, considering that this required approximately 80% of the layout time for the networks we considered. Yet, doing so limits the scalability of our approach as the number of GPUs is increased or if denser networks, with relatively many edges, are considered. As such we suggest future research to

consider distributing additional components of the layout algorithm between different GPUs.

  - In our implementation of the multi-GPU network renderer, we distribute the render work between different GPUs through a spatial partitioning of the layout space. In the context of tiled display visualization, this has the advantage that each GPU can render the graphics to be displayed on attached monitors, reducing inter-GPU data transfers. However, as our results demonstrated, this work distribution does not necessarily ensure a uniform load balancing between different GPUs. We would suggest future research to evaluate the trade-off between the cost of inter-GPU data transfers and the benefits of a balanced workload between GPUs. Initially this could amount to distributing geometrical primitives, i.e. the lines for edges and circles for nodes, uniformly between GPUs, rather than through a segmentation of the layout space.
  - Future research could also consider using a dedicated GPU server (cluster) to compute and render network drawings. We expect ensuring real-time, low-latency, interactivity is most challenging when using this approach. For example, to provide the 25 megapixel tiled display system used in this study with video frames at the 60 Hz. refresh rate of the monitors, would result in a data stream of approximately 47 Gbit/s from the server to the tiled display system. Facilitating this, through high-speed interconnects and system I/O, or reducing the stream's data rate, possibly through compression, would be challenges posed by this approach. Since the code developed for this study supports server-based OpenGL network rendering without requiring a window system, by means of the Khronos EGL API [28], it could serve as a starting point.

- With the purpose of evaluating the interactive capabilities of our system, we implemented a number of interactions, enabling users of the system to navigate the network and to locally destabilize it through the 'local repulse' and 'local heat' modes. There are various other interactions that we did not consider here, which would yield a significant improvement in the system's data-exploration capabilities, based on our experience with the system and users' comments. Especially useful would be means for the user to obtain properties of selected nodes and edges in the networks, for example through pop-up menu's or node labels providing information on selected nodes, or by highlighting the nodes and edges connected to a selected node. The possibility to filter for nodes or edges with certain properties, or to color and size them based on these properties, would also be valuable for the purposes of data-exploration. To improve users' sense of context, i.e. what part of the drawing they are viewing, a map of the network drawing indicating the part that is viewed could be superimposed in one of the display's corners.

- The tiled visualization approach we employed has to synchronize updates to the different monitors in the system to ensure a coherent image spans all monitors. To this end we synchronize the initiation of the OpenGL buffer swap procedure for different GPUs, on the CPU. As discussed this is a sub-optimal approach, since it synchronizes the initiation of the buffer swap process rather than the actual buffer swap. Given the importance of image coherency we deem improvements in this area valuable.

Chapter 8

Conclusion

In this thesis we presented an approach to using the graphics processing units (GPUs) in a tiled display system for interactive network visualization at high resolutions. We hypothesized that using the GPU as a platform for both network layout and network drawing would allow for high performance levels, enabling real-time interaction for networks with hundreds of thousands of nodes and edges. We evaluated this hypothesis using a visualization approach that incorporates multi-GPU force-directed graph layout and a distributed rendering approach in which each GPU in the system draws the part of the network to be displayed on the monitors attached to it. An evaluation of our approach demonstrated real-time performance at 60 frames per second (fps) for networks with tens of thousands of nodes and edges, on a 25 megapixel tiled display system with twelve monitors and three NVIDIA GeForce GTX660 graphics cards. This constitutes a performance improvement of 3.9× over the standard single GPU implementation that served as the starting point for our multi-GPU approach. We were able to successfully implement real-time navigation and a number of interactions with the layouts using a Nintendo Wii remote as input device. Our results are promising, and present three main challenges to realize further improvements in future work. First, the performance of our multi-GPU network visualization approach appears to be limited due to the time spent on repulsive force approximation, due to a number of non-distributed components in our implementation of the layout algorithm, and potentially also due to the approach used for distributed rendering. Second, additional interactions need to be realized to facilitate effective data-analysis and exploration. Finally, we need to be able to enforce synchronous updates between the different monitors in the tiled display system. This would resolve the remaining discontinuities that currently appear, especially if frame rates decrease as result of increased network sizes. We believe these problems can be resolved without a major revision of the approach presented in this study, and as such we expect future work to be effective at scaling the approach to networks with hundreds of thousands of nodes and edges.

Acknowledgment

I would like to thank Kristian Rietveld and Fons Verbeek, for their encouragement and advice during this study. The (often technical) discussions with Kristian Rietveld were always insightful, motivating and introduced me to numerous interesting topics. Similarly, the suggestions, support and enthusiasm by Fons Verbeek resulted in improvements to many different aspects of this work. Thank you both for the pleasant collaboration on this project.

Appendix A

Network Properties

Table A.1 provides a number of properties for the network data used in this thesis. Most data was obtained from the KONECT [33] and SNAP [35] repositories.

Network	Nodes	Edges	Density	Avg. Degree
Amazon0505	410,236	2,439,437	0.0000	11.9
Brightkite	56,739	212,945	0.0001	7.5
CA-GrQc	4,158	13,422	0.0016	6.5
Cit-HepPh	34,401	420,784	0.0007	24.5
Cit-HepTh	27,400	352,021	0.0009	25.7
GoogleNw	15,763	148,585	0.0012	18.9
Newman-Cond_mat	22,015	58,578	0.0002	5.3
PGPgiantcompo	10,680	24,316	0.0004	4.6
auto	448,695	3,314,611	0.0000	14.8
ca-AstroPh	17,903	196,972	0.0012	22.0
ca-CondMat	21,363	91,286	0.0004	8.5
ca-HepPh	11,204	117,619	0.0019	21.0
ca-HepTh	8,638	24,806	0.0007	5.7
cnr_2000	325,557	2,738,969	0.0001	16.8
com-amazon	334,863	925,872	0.0000	5.5
dblp20080824	511,163	1,871,070	0.0000	7.3
dip	19,928	41,202	0.0002	4.1
email-Enron	33,696	180,811	0.0003	10.7
email-EuAll	224,832	339,924	0.0000	3.0
orkut	3,072,441	117,185,083	0.0000	76.3
p2p-Gnutella31	62,561	147,877	0.0001	4.7
petster	1,788	12,475	0.0078	14.0
ppi_dip_swiss	3,766	11,922	0.0017	6.3
ppi	37,333	135,618	0.0002	7.3
soc-Epinions1	75,877	405,738	0.0001	10.7
soc-Slashdot0902	82,168	504,230	0.0001	12.3
wiki-Vote	7,066	100,735	0.0040	28.5
wiki	1,791,489	25,444,207	0.0000	28.4
ydata-ysm-ad	653,260	2,278,448	0.0000	7.0
youtube	1,134,890	2,987,624	0.0000	5.3

Table A.1: Additional properties for the network data used in this thesis. Original data was filtered for the LWCC, self-loops were removed and the resulting graph was considered to be undirected. All properties were computed using NetworkX (version 2.1) [24].

Appendix B

Additional Results

Dataset	Force Approximation	Tree Summarize	Tree Building
petster	0.91 (77.41%)	0.14 (11.88%)	0.04 (3.36%)
ppi_dip_swiss	1.24 (76.08%)	0.24 (14.83%)	0.05 (3.00%)
CA-GrQc	1.26 (75.95%)	0.25 (15.21%)	0.05 (2.90%)
wiki-Vote	2.38 (74.94%)	0.49 (15.27%)	0.07 (2.19%)
ca-HepTh	2.83 (80.06%)	0.48 (13.55%)	0.09 (2.58%)
PGPgiantcompo	2.80 (77.33%)	0.58 (15.93%)	0.10 (2.67%)
ca-HepPh	3.45 (76.93%)	0.68 (15.11%)	0.10 (2.28%)
GoogleNw	4.88 (78.86%)	0.91 (14.63%)	0.13 (2.06%)
ca-AstroPh	5.50 (78.34%)	0.99 (14.15%)	0.14 (1.94%)
dip	6.20 (81.98%)	1.00 (13.19%)	0.14 (1.84%)
ca-CondMat	6.55 (80.99%)	1.12 (13.84%)	0.15 (1.80%)
Newman-Cond_mat	7.96 (84.59%)	1.04 (11.10%)	0.14 (1.54%)
Cit-HepTh	9.06 (79.70%)	1.47 (12.90%)	0.18 (1.56%)
email-Enron	9.62 (78.72%)	1.92 (15.74%)	0.21 (1.71%)
ppi	11.20 (80.44%)	2.05 (14.74%)	0.23 (1.62%)
Cit-HepPh	11.40 (81.44%)	1.62 (11.55%)	0.20 (1.45%)
Brightkite	18.20 (81.41%)	3.13 (13.98%)	0.34 (1.50%)
p2p-Gnutella31	19.39 (81.47%)	3.29 (13.84%)	0.36 (1.51%)
soc-Epinions1	24.37 (80.00%)	4.39 (14.43%)	0.46 (1.51%)
soc-Slashdot0902	27.94 (79.62%)	4.90 (13.96%)	0.50 (1.44%)
email-EuAll	86.19 (83.79%)	12.70 (12.35%)	1.63 (1.59%)
Average	79.53%	13.91%	2.00%

Table B.1: Average amount of time (in ms) spent on the three most time consuming components of the ForceAtlas2 algorithm, during a single iteration. Fractions, relative to all components, are given between braces. Results are averaged over the first 500 iterations of the algorithm and obtained using a single GPU on BigEye.

Dataset	Force Approximation	Attractive Force	Tree Building
com-amazon	15.65 (88.02%)	0.41 (2.33%)	0.66 (3.73%)
cnr_2000	14.91 (81.42%)	1.72 (9.41%)	0.66 (3.61%)
Amazon0505	20.35 (85.93%)	1.15 (4.87%)	0.83 (3.51%)
auto	22.36 (85.39%)	1.42 (5.44%)	0.88 (3.34%)
dblp20080824	24.76 (86.12%)	1.08 (3.75%)	1.07 (3.72%)
ydata-ysm-ad	52.23 (85.85%)	4.12 (6.77%)	1.49 (2.45%)
youtube	110.85 (90.50%)	3.02 (2.47%)	3.03 (2.47%)
wiki	222.16 (79.25%)	43.27 (15.44%)	5.44 (1.94%)
orkut	497.25 (63.71%)	255.16 (32.69%)	10.71 (1.37%)
Average	82.91%	9.24%	2.91%

Table B.2: Average amount of time (in ms) spent on the three most time consuming components of the ForceAtlas2 algorithm, during a single iteration. Fractions, relative to all components, are given between braces. Results are averaged over the first 500 iterations of the algorithm and obtained using a single GPU for the datasets and GPU used during the scalability experiments

Dataset	Force Approximation	Tree Summarize	Tree Building
petster	0.87 (76.25%)	0.14 (12.12%)	0.04 (3.42%)
ppi_dip_swiss	1.17 (74.32%)	0.24 (15.35%)	0.05 (3.12%)
CA-GrQc	1.19 (74.18%)	0.25 (15.74%)	0.05 (3.03%)
wiki-Vote	1.36 (62.35%)	0.49 (22.21%)	0.07 (3.17%)
ca-HepTh	1.49 (66.68%)	0.49 (21.85%)	0.09 (4.12%)
PGPgiantcompo	1.48 (63.29%)	0.58 (24.78%)	0.10 (4.17%)
ca-HepPh	1.87 (63.71%)	0.68 (22.96%)	0.10 (3.48%)
GoogleNw	2.94 (68.63%)	0.90 (20.90%)	0.13 (2.96%)
dip	3.38 (70.36%)	1.00 (20.76%)	0.14 (2.91%)
ca-AstroPh	3.28 (67.56%)	0.99 (20.48%)	0.14 (2.81%)
ca-CondMat	3.34 (67.70%)	1.11 (22.51%)	0.15 (2.94%)
Newman-Cond_mat	4.06 (72.95%)	1.04 (18.62%)	0.14 (2.59%)
Cit-HepTh	5.13 (68.17%)	1.47 (19.54%)	0.18 (2.36%)
email-Enron	4.98 (64.77%)	1.93 (25.09%)	0.21 (2.73%)
Cit-HepPh	5.86 (68.47%)	1.61 (18.87%)	0.20 (2.39%)
ppi	5.92 (67.62%)	2.05 (23.45%)	0.22 (2.56%)
Brightkite	9.73 (69.18%)	3.13 (22.27%)	0.34 (2.40%)
p2p-Gnutella31	9.80 (68.05%)	3.30 (22.90%)	0.36 (2.50%)
soc-Epinions1	12.34 (66.15%)	4.39 (23.54%)	0.46 (2.48%)
soc-Slashdot0902	14.58 (66.31%)	4.89 (22.24%)	0.51 (2.30%)
email-EuAll	42.85 (71.49%)	12.48 (20.82%)	1.60 (2.66%)
Average	68.49%	20.81%	2.91%

Table B.3: Average amount of time (in ms) spent on the three most time consuming components of the ForceAtlas2 algorithm, during a single iteration. Fractions, relative to all components, are given between braces. Results are averaged over the first 500 iterations of the algorithm and obtained using two GPUs on BigEye.

Bibliography

[1] Josh Barnes and Pete Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the Third International Conference on Weblogs and Social Media*, 2009.

[3] Jeroen Bédorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 54–65. IEEE Press, 2014.

[4] Thijs Beuming, Lucy Skrabanek, Masha Y. Niv, Piali Mukherjee, and Harel Weinstein. Pdzbase: a protein-protein interaction database for pdz-domains. *Bioinformatics*, 21(6):827–828, 2005.

[5] David Blythe. Rise of the graphics processor. In *Proceedings Of The IEEE*, pages 761–778, 2008.

[6] OpenMP Architecture Review Board. Openmp application program interface, version 4.0 - july 2013. `http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`, 2013. Accessed: 30-07-2018.

[7] Govert G. Brinkmann. Project page - interactive visualization of large networks on a tiled display system. `https://govertbrinkmann.nl/mthesis`. Accessed: 30-07-2018.

[8] Govert G. Brinkmann, Kristian F. D. Rietveld, and Frank W. Takes. Exploiting gpus for fast force-directed visualization of large-scale networks. In *46th International Conference on Parallel Processing*, pages 382–391, 2017.

[9] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In Wen mei W. Hwu, editor, *GPU Computing Gems Emerald Edition*, chapter 6, pages 75–92. Morgan Kaufmann, 2011.

[10] Sangwon Chae. *HD-GraphViz: Highly Distributed Graph Visualization on Tiled Displays.* PhD thesis, University of California, Irvine, 2013.

[11] Sangwon Chae, Aditi Majumder, and M. Gopi. Hd-graphviz: highly distributed graph visualization on tiled displays. In *The Eighth Indian Conference on Vision, Graphics and Image Processing, ICVGIP '12, Mumbai, India, December 16-19, 2012*, page 43, 2012.

[12] Adrien Douady. *Julia Sets and the Mandelbrot Set*, pages 161–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.

[13] CORPORATE The MPI Forum. MPI: a message passing interface. In *Proceedings Supercomputing '93, Portland, Oregon, USA, November 15-19, 1993*, pages 878–883, 1993.

[14] Yaniv Frishman and Ayellet Tal. Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

[15] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience.*, 21(11):1129–1164, 1991.

[16] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[17] James Gettys, Robert W. Scheifler, Chuck Adams, Vania Joloboff, Hideki Hiura, Bill McMahon, Ron Newman, Al Tabayoyon, Glenn Widener, and Shigeru Yamada. Xlib - c language x interface. `https://www.x.org/releases/X11R7.7/doc/libX11/libX11/libX11.pdf`, 2002. Accessed: 30-07-2018.

[18] Jim Gettys and Keith Packard. The x resize, rotate and reflect extension. `https://cgit.freedesktop.org/xorg/proto/randrproto/tree/randrproto.txt`, 2015. Accessed 16-03-2018.

[19] Helen Gibson, Joe Faith, and Paul Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization*, 12(3-4):324–357, 2013.

[20] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C. Hart. Rapid multipole graph drawing on the GPU. In *Graph Drawing, 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers*, pages 90–101, 2008.

[21] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C. Hart. Rapid multipole graph drawing on the GPU. In *Graph Drawing, 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers*, pages 90–101, 2008.

[22] Khronos OpenCL Working Group. Opencl specification. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf`, 2017. Accessed: 30-07-2018.

[23] Yi Gu, Chaoli Wang, Jun Ma, Robert J. Nemiroff, and David L. Kao. igraph: a graph-based technique for visual analytics of image and text collections. In *Visualization and Data Analysis 2015, San Francisco, CA, USA, February 9-11, 2015*, page 939708, 2015.

[24] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11—-15, 2008.

[25] David Herrmann. Xwiimote - open-source nintendo wii / wii u device driver. `https://dvdhrm.github.io/xwiimote/`. Accessed: 30-07-2018.

[26] Yifan Hu and Lei Shi. Visualizing large graphs. *Wiley Interdisciplinary Reviews: Computational Statistics*, 7(2):115–136, 2015.

[27] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 693–702, 2002.

[28] The Khronos Group Inc. Khronos native platform graphics interface. https://www.khronos.org/registry/EGL/specs/eglspec.1.4.pdf, 2013. Accessed: 30-07-2018.

[29] Intel Corporation. Intel(r) 64 and ia-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm, 2018. Accessed: 30-07-2018.

[30] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. *PLoS ONE*, 9(6):1–12, 2014.

[31] Ryoichi Jingai, Yoshiyuki Kido, Susumu Date, and Shinji Shimojo. Research note: A high resolution graph viewer for multi-monitor visualization environment. *The Review of Socionetwork Strategies*, 9(1):19–27, Jun 2015.

[32] KONECT. Pdzbase network dataset – konect, april 2017. http://konect.uni-koblenz.de/networks/maayan-pdzbase. Accessed: 30-07-2018.

[33] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proceedings WWW*, pages 1343–1350, 2013.

[34] Jason Leigh, Andrew E. Johnson, Luc Renambot, Tom Peterka, Byungil Jeong, Daniel J. Sandin, Jonas Talandis, Ratko Jagodic, Sungwon Nam, Hyejung Hur, and Yiwen Sun. Scalable resolution display walls. *Proceedings of the IEEE*, 101(1):115–129, 2013.

[35] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. https://snap.stanford.edu/data, June 2014. Accessed: 30-07-2018.

[36] Peng Mi, Maoyuan Sun, Moeti Masiane, Yong Cao, and Chris North. Interactive graph layout of a million nodes. *Informatics*, 3(4):23, 2016.

[37] Christopher Mueller, Douglas P. Gregor, and Andrew Lumsdaine. Distributed force-directed graph layout and visualization. In *Eurographics Symposium on Parallel Graphics and Visualization, EGPGV 2006, Braga, Portugal*, pages 83–90, 2006.

[38] Govind Mukundan. Drawing anti-aliased circular points using opengl/webgl - a circular reference. https://www.desultoryquest.com/blog/drawing-anti-aliased-circular-points-using-opengl-slash-webgl/.

[39] NVIDIA Corporation. Appendix b. x config options. https://download.nvidia.com/XFree86/Linux-x86_64/390.42/README/xconfigoptions.html. Accessed: 30-07-2018.

[40] NVIDIA Corporation. Chapter 11. specifying opengl environment variable settings. https://download.nvidia.com/XFree86/Linux-x86_64/390.42/README/openglenvvariables.html. Accessed: 30-07-2018.

[41] NVIDIA Corporation. Chapter 12. configuring multiple display devices on one x screen. https://download.nvidia.com/XFree86/Linux-x86_64/390.42/README/configtwinview.html. Accessed: 30-07-2018.

[42] NVIDIA Corporation. Chapter 32. offloading graphics display with randr 1.4. https://download.nvidia.com/XFree86/Linux-x86_64/390.42/README/randr14.html. Accessed: 30-07-2018.

[43] NVIDIA Corporation. Cuda toolkit documentation. https://docs.nvidia.com/cuda/. Accessed: 30-07-2018.

[44] NVIDIA Corporation. Github - nvidia/nvidia-settings: Nvidia driver control panel. https://github.com/NVIDIA/nvidia-settings. Accessed: 30-07-2018.

[45] NVIDIA Corporation. Profiler user's guide. https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf. Accessed: 30-07-2018.

[46] NVIDIA Corporation. Programming guide :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#graphics-interoperability. Accessed: 30-07-2018.

[47] NVIDIA Corporation. Glx_nv_swap_group specification. https://www.khronos.org/registry/OpenGL/extensions/NV/GLX_NV_swap_group.txt, 2008. Accessed: 18-03-2018.

[48] NVIDIA Corporation. Ext_swap_control specification. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_swap_control.txt, 2011. Accessed: 18-06-2018.

[49] Robert W. Scheifler. X window system protocol x consortium standard x version 11, release 6.7. https://www.x.org/docs/XProtocol/proto.pdf, 2004. Accessed: 30-07-2018.

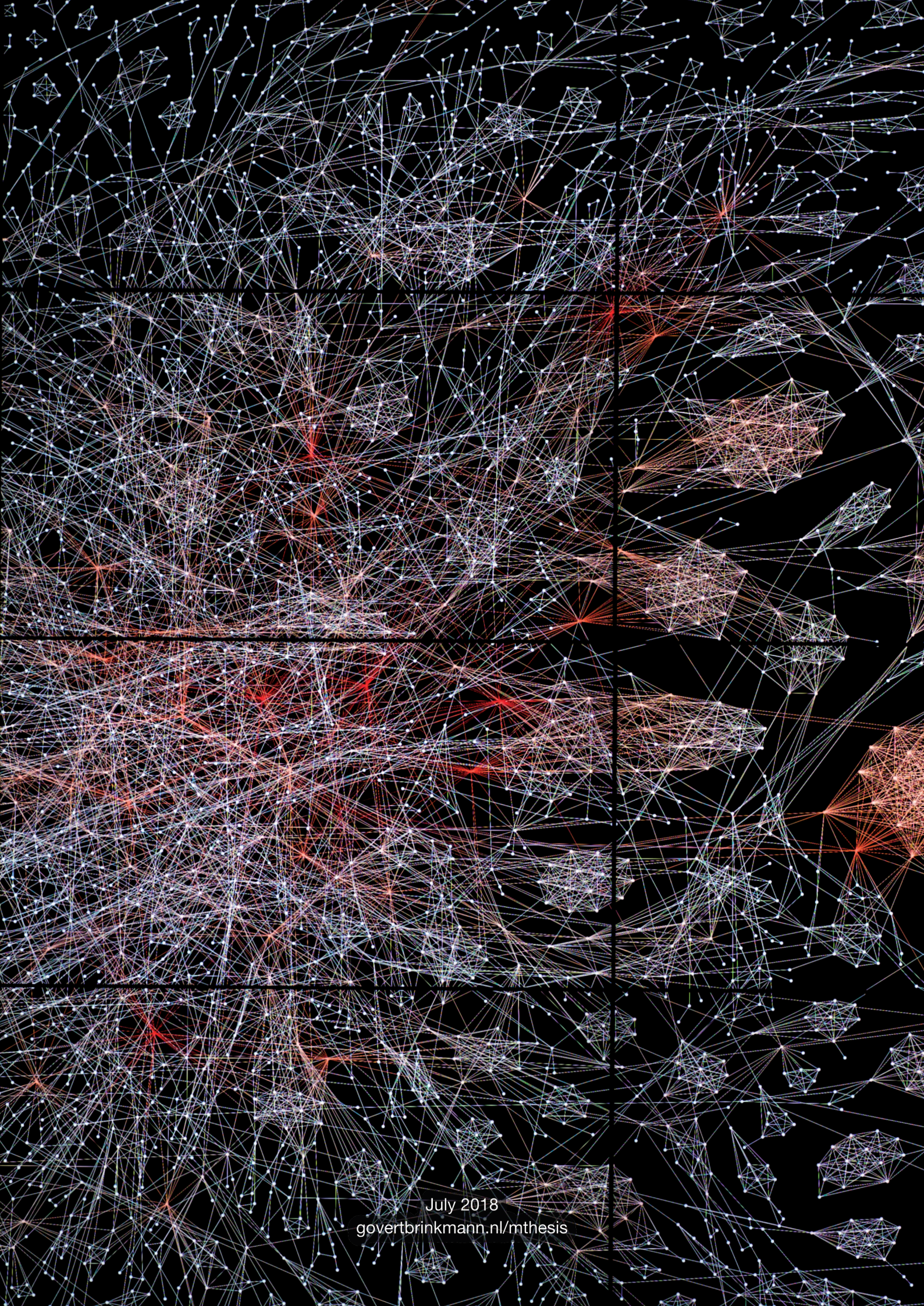
[50] Mark Segal and Kurt Akeley. The opengl graphics system: A specification. https://www.khronos.org/registry/OpenGL/specs/gl/glspec41.core.pdf, 2017. Accessed: 30-07-2018.

[51] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, USA, September 3-6, 1996*, pages 336–343, 1996.

[52] Frank W. Takes and Eelke M. Heemskerk. Centrality in the global network of corporate control. *Social Network Analysis and Mining*, 6(1):97:1–97:18, 2016.

[53] Chris Wellons. Mandelbrot set with simd intrinsics. https://nullprogram.com/blog/2015/07/10/, 2015. Accessed: 30-07-2018.

[54] xcb.freedesktop.org. The x protocol c-language binding. Accessed: 30-07-2018.



July 2018
govertbrinkmann.nl/mthesis