GPU Based Methods for Interactive Information Visualization of Big Data

Peng Mi

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Christopher L. North, Chair
Yong Cao
Denis Gračanin

Dec 9, 2015
Blacksburg, Virginia

Keywords: GPU based Algorithm, Information Visualization

GPU Based Methods for Interactive Information Visualization of Big Data

Peng Mi

(ABSTRACT)

Interactive visual analysis has been a key component of gaining insights in information visualization area. However, the amount of data has increased exponentially in the past few years. Existing information visualization techniques lack scalability to deal with big data, such as graphs with millions of nodes, or millions of multidimensional data records.

Recently, the remarkable development of Graphics Processing Unit (GPU) makes GPU useful for general-purpose computation. Researchers have proposed GPU based solutions for visualizing big data in graphics and scientific visualization areas. However, GPU based big data solutions in information visualization area are not well investigated. In this thesis, I concentrate on the visualization of big data in information visualization area. More specifically, I focus on visual exploration of large graphs and multidimensional datasets based on the GPU technology. My work demonstrates that GPU based methods are useful for sensemaking of big data in information visualization area.

Acknowledgment

Firstly, I would like to express my sincere gratitude to my advisors, Professor Christopher L. North and Yong Cao, for supporting me throughout my graduate study. Dr. North is the funniest advisor and one of the smartest people I know. His questions are always insightful and give me new angles towards my work. Dr. Cao guides me to the GPU world, and encourages me to enter the information visualization area. Without his support, my work would not have become reality. I am also very grateful to Dr. Denis Gračanin for his insightful discussions and suggestions about my work.

I would like to acknowledge Dr. Pak Chung Wong and Kris Cook at Pacific Northwest National Laboratory (PNNL). I am thankful to Kris Cook who gives me an opportunity as a summer intern at PNNL. Dr. Wong is my advisor at PNNL, who helps me improve my knowledge in the area.

Finally, I appreciate the financial support from PNNL that funded parts of the research discussed in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Big data is everywhere – from sensors that monitor network flows to the flood of social networks, we are surrounded by massive digital content. Big data encompasses everything, which can unleash new values and insights, and potentially supports people new capabilities for decision-making [12].

However, there is a wide gap between the big data and its insights [43]. The data volume and complexity impede progress for sensemaking of big data. As available data becomes more extensive and complex, the visualization based data discovery can efficiently and effectively deliver insights from big data. However, weaving big data into interactive visualizations that provides understanding and sense-making is a big challenge.

Liu et al. [45] discussed various techniques that enable interactive visualization of big data, where real-time performance is one of the fundamental requirements. Their paper highlights GPU based solutions for quickly data filtering to achieve real-time frame-rates. Therefore, designing and implementing GPU based big data visualization systems become a trend. In this chapter, I would like to present the motivations behind GPU based interactive information visualization of big data.

1.1 Research Motivations

1.1.1 The Interactivity of Big Data

We have entered in the era of "big data". Data grows at unprecedented speed. Most of them are network logs, social networks and financial transactions, which belong to data domain of information visualization. However, current information visualization techniques focus on small data (less than one million data items). Thus how to extend interactive information visualization techniques to big data becomes my thesis topic.

Figure 1.1 describes the data scalability from small data to big data. In this thesis, I focus on extending the interactivity of information visualization from small data to big data, where data items can reach beyond one million.

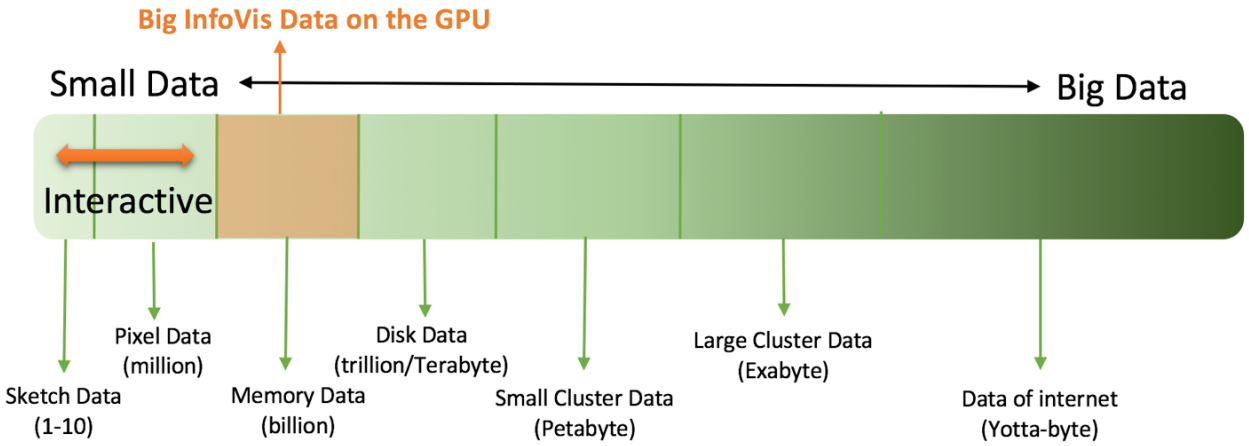

Figure 1.1: The data scalability from small data to big data.

Therefore, the first motivation of this thesis is interactively visualizing big data in information visualization area. The number of data items should be larger than one million, and the data can be stored in the GPU memory (upper bound).

1.1.2 The GPU Acceleration

GPUs are designed for graphics and videogames. In recently years, the GPU has been praised for the significant performance improvement and the capability of general purpose computation. GPUs have been widely used in various domains (e.g., data mining [17], machine learning [46], high performance computing [16], etc.).

The GPU based visualization techniques have been investigated, which mostly are about volume data visualizations [4] in scientific visualization (SciVis) area. The GPU based InfoVis solutions are not investigated well. Therefore, designing GPU based solutions in InfoVis area by utilizing GPU fine-grained parallelism and high memory bandwidth is the second motivation of this thesis.

1.2 Contributions

In my thesis, I focus on two common data types in the data domain of InfoVis: graphs and multidimensional datasets. And there are two key contributions towards the state-of-arts in visual exploration of graphs and multidimensional datasets:

1. **Large graph layout on the GPU.** I proposed a topology oriented force-directed graph layout algorithm to calculate the approximate repulsive force for large graph layout. I designed GPU friendly data structures and GPU parallel algorithms to improve graph layout performance. As a result, my parallel algorithm can reach real-time frame-rates for visualizing graphs with a million nodes.

2. **Visual exploration of large multidimensional datasets on the GPU.** To achieve flexible filtering and gaining fine-grained data details, I proposed a GPU-centric design to emphasize that data storage and computation are on the GPU. I designed a data dependency graph for characterizing the data flow on the GPU. I implemented an Animated VISualization Tool - AVIST, as a proof-of-concept, for visually exploring millions data records.

1.3 Thesis Organization

The reminder of this thesis is organized as follows. Chapter 2 presents a topology based force-directed graph layout on the GPU. This algorithm enables human interactions with automatic graph layouts. Chapter 3 focuses on the exploratory visual analysis of large multidimensional datasets. It describes an exploration oriented tool - AVIST, which can support visual analysis of millions of data records. Chapter 4 gives the conclusion and proposes future work.

Chapter 2

Human Interaction with Large Graph Layout on the GPU

2.1 Introduction

Graphs are commonly used to depict complex relationships among objects. Graph drawing offers solutions to geometrically represent graphs, with the intention of improving their readability. This supports applications and analysis in various domains (e.g., social network analysis [31, 33], cyber security [69], intelligence analysis [19, 59], etc.).

The booming of information technology brings in data as a deluge, which leads to a rapid growth of data in size and complexity. Although a graph layout provides a human perceptible way to support sensemaking tasks [33, 58], the performance of drawing large and complex graphs, especially those with millions of nodes, is still challenging. Many graph drawing algorithms have been proposed in the last few decades. However, most of them work fine for relatively small graphs (e.g., graphs with hundreds of nodes) and certain types of graphs (e.g., trees or planar graphs) [39, 61]. Some existing large graph layout algorithms [27] focus on generating static and visual pleasing results, but layout results from these are constrained by their predefined aims. Therefore, those layout algorithms cannot support users making sense of large graphs in an efficient and effective manner.

In this chapter, we argue that *direct manipulations of large graphs and real-time interactions are vital to support knowledge discovery and sensemaking activities for large graphs*. Previous graph layout algorithms emphasize static graph layout results either based on their structures [27] or semantic meanings [55]. However, a static graph layout may not reveal all information of a graph. In some cases, users have no clear definition of the best graph layout. They need to explore a graph, and interact with its layout. Thus, users can incrementally update a graph layout, and emphasize different graph aspects for gaining different insights. In this chapter, we highlight the interaction for a large graph layout, which potentially supports users exploring unexpected knowledge of a large graph. We stress that the graph layout performance should be single iteration oriented for real-time human interactions.

To achieve this, we focus on a classical force-direct algorithm using the spring-electrical model [21]. This algorithm depicts the graph drawing problem as a physical system of spring-like attractive forces generated by each edge, while each charged node repels others via electrical force. Since it iteratively calculates each node position, this algorithm has the inherent ability to enable incorporating user interactions into a automatic layout process. In each iteration, users can flexibly choose certain nodes, drag and pin them to modify layout results. However, this algorithm does not scale well with respect to the size of graphs, and it suffers from poor scalability even on moderate-sized graphs.

Similar to the N-body problem, the performance bottleneck of this algorithm lies in repulsive force calculation [25]. Many existing solutions have been proposed to replace computing the exact repulsive forces between all pairs of nodes with some approximated calculations. The heuristic is that if two nodes are far away from each other, the repulsive force between them can be ignored or approximated. Thus, a spatial indexing structure should be built and updated to describe the spatial relationships among nodes in each iteration. For example, Godiyal et al. [23] proposed a method combing CPU and GPU to construct a balance k-d tree, while Martin et al. [7] gave a GPU based octree implementation. However, building and traversing hierarchical structures can be computationally expensive for large graphs.

In this chapter, we propose a new solution to speed up repulsive force calculation to enable human-in-the-loop for large graph layout. We calculate approximate repulsive forces based

on a graph structure, rather than its nodes' spatial distribution. The benefit of our algorithm, compared with previous accelerated algorithms, is that we avoid building and traversing a spatial indexing structure. To further improve performance, we design a novel parallel force-directed graph layout algorithm that utilizes the massive computational power of GPUs to achieve real-time frame-rates so as to support human interactions. In addition, our algorithm can be integrated into the multi-level graph layout paradigm, which solves the local minimum problem to generate visual pleasing results.

The reminder of this chapter is organized as follows: In Section 2, we review relevant work about large graph layouts. In Section 3, we describe our approximated force-directed graph layout algorithm, and its integration with the multi-level paradigm. Section 4 addresses the details of GPU implementation. Performance evaluation of our algorithm and visual assessment analysis are discussed in Section 5. In Section 6, we demonstrate two case studies to emphasize human-in-the-loop of exploring graph layouts. We summarize lessons learned for designing big graph layout algorithms in Section 7. Finally, we conclude this chapter in Section 8.

2.2 Related Work

In this section, we review large graph layout algorithms. In addition, we focus on force-directed algorithms and GPU based methods.

2.2.1 Large Graph Layout

Hachul et al. [27] surveyed existing solutions that generate a large graph layout by considering two aspects: aesthetics and performance. They suggested that certain force-directed layout algorithms can generate pleasing layouts for most tested graphs. Several recent papers [20, 23, 34, 60, 67] presented their advanced large graph layout techniques based on force-directed algorithms [13, 14, 21, 38].

Another strategy for a large graph layout is based on linear algebra, rather than physical

simulation, such as HDE [28] and ACE [42]. However, these algorithms work fine on few specific graphs. Moreover, linear algebra based methods cannot be easily applied to integrate human interactions.

In addition to the algorithms mentioned above, other large graph drawing algorithms have been investigated. For example, Muelder et al. [50, 49] presented large graph layout algorithms based on treemaps; Wong et al. [66] used a space-filling fractal curve to layout graph nodes; Khoury et al. [40] gave a layout algorithm that simplifies matrix computations based on linear-algebraic properties. However, these works focus on the generation of static visual representations of large graphs, so they are hard to interactively change their graph layouts for analytical purposes (e.g., pin some interesting nodes and drag certain parts of the graph).

2.2.2 Force-Directed Algorithms

Force-directed algorithms are commonly used to support visual analysis of graphs, because they are conceptually simple and able to produce aesthetically pleasing layouts. These algorithms are also called energy-based methods, indicating that the minimum energy is achieved when the net force on every vertex is zero, and when a graph layout is generated. Many practical algorithms are proposed, for more details one can refer to [41].

The spring-electrical model [21] is a popular force-directed algorithm, which generates a graph layout based on two types of forces: attractive force and repulsive force. This algorithm cannot handle a large graph layout well. Its complexity of calculating repulsive force is $O(N^2)$, where N denotes number of nodes in a graph. For a graph with a million nodes, each iteration of repulsive force calculation needs tera-sized computations, which is beyond the processing power of a single commodity desktop. To speed up its performance, a natural approach is to compute approximated force based on some heuristic method instead of striving for the optimal solution. Fruchterman et al. [21] proposed a grid-variant algorithm that accelerates repulsive force computation by splitting nodes into grids. Based on the Barnes-Hut Tree [5], a tree structure is used to speed up repulsive force calculation by grouping far away vertices as supernodes ([34], [53]). Yunis et al. [67] and Godiyal et al. [23] proposed

Fast Multiple Method (FMM) [25] for accelerating repulsive force computation.

Besides high performance, another problem with spring-electrical model is the local minimum configuration of a large graph layout, especially when the beginning graph layout is from a random initialization. A multi-level approach can overcome this limitation [22, 34, 62]: a sequence of successive smaller graphs are generated by graph coarsening and partitioning techniques to simplify the topology of its parent. Global optimal layout is obtained from a small graph, which is then used as a starting layout for the next level, until the finest graph layout has been achieved. Gereon Bartel et al. [6] summarized the multi-level layout paradigm in three phases: coarsening, placement and single level layout. They concluded that there is no clear winning combination, since different coarsening, placement and layout algorithms have different attributes.

2.2.3 Graph Layout on the GPU

GPUs are designed for videogames and graphics. The remarkable advances in performance and programmability make GPUs popular for general purpose computation. Frishman et al. [20] made the first step towards GPU based graph layout algorithms. Apeksha et al. [23] presented a parallel FMM algorithm on the GPU, and used a k-d tree structure for describing nodes' spatial distribution. Yunis et al. [67] extended a parallel FMM algorithm to multiple GPUs. Besides, Auber et al. [3] and Jeowicz et al. [37] also presented their GPU accelerated solutions to speed up graph layouts. Moreover, Tikhonova et al. [60] proposed a scalable parallel force-directed layout algorithm in parallel and distributed computing environments.

In this chapter, we aim at applying the spring-electrical model to a large graph layout, wherein users can interact with the graph layout to perceive and explore more meaningful information. Previous large graph layout algorithms cannot easily achieve this. To the best of our knowledge, existing acceleration techniques cannot achieve fast performance for large graph layout (specifically those with millions of nodes) to support some exploratory interactions. We propose a GPU based approximated force-directed layout algorithm, which can support real-time human interactions on the large graph mentioned above. This potentially

supports real-time explorations of these large graphs in future.

2.3 Algorithm Outline

In this section, we describe our algorithm design, which addresses the problems of spring-electrical model for a large graph layout from two key aspects: performance and quality results.

2.3.1 Approximation of The Repulsive Force

Both Barnes-Hut Tree [5] and FMM [25] methods calculate approximated repulsive force based on nodes' distribution and a spatial indexing structure that should be built and updated in each iteration. To avoid these stages, we propose to use a graph topology such as heuristic to approximately calculate repulsive forces. If two nodes belong to two different graph clusters, they are "far away" from each other, and we use their inter-cluster repulsive force for approximation. In total, repulsive forces have two components: *internal-electric-force* and *external-electric-force*. The *internal-electric-force* refers to the repulsive force between pairs of nodes within the same cluster, and the *external-electric-force* is the inter-cluster repulsive force. The *internal-electric-force* is used to obtain the local structure of a graph, while the *external-electric-force* is used for capturing the overview of a graph.

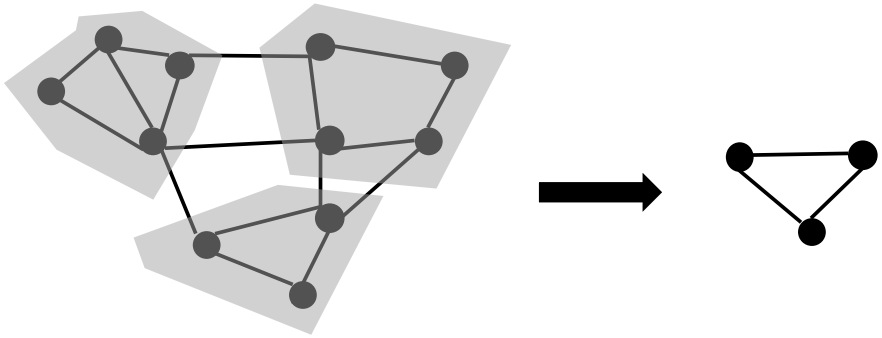

Figure 2.1: The original graph (left) and its coarsen graph (right).

Given an undirected graph $G = G^0 = (V, E)$, firstly we use a multi-level coarsening or

clustering algorithm to generate serial successively coarser graphs $(G^1, G^2, \cdots)$, where each node in the upper level represents a cluster of nodes in the lower level. Figure 2.1 shows two level graphs, and the original graph is coarsened based on its topology structure. To get each node's repulsive force in level i with graph G_i, we just need to calculate the *internal-electric-force* , while the *external-electric-force* is the repulsive force of graph G_{i-1}. We calculate *internal-electric-force* in each level of the graph and pass repulsive forces to the next level as *external-electric-force* until we reach the finest level of the graph. Finally, we summarize the attractive force and the approximated repulsive force of the finest level graph and update each node's position.

Assuming that the total number of nodes is N, a graph is evenly partitioned into clusters with P nodes. We have N/P number of clusters in the finest level, which equals to the number of nodes in its upper level. The new level, then has N/P^2 clusters. In summary, the total number of clusters is $N/P + N/P^2 + N/P^3 + \cdots = N/(P-1)$. The complexity of *internal-electric* is $O(P^2)$, so the total repulsive force complexity is $O(NP)$. In addition to the attractive force, the total complexity is $O(NP + E) \approx O(N + E)$, where E is the number of edges in the graph, and $N >> P$.

Actually, a graph cannot be evenly partitioned based on its topology, such as a scale-free network, where its nodes' distribution follows a power law. Thus, the value of P varies for different clusters. We may constrain the size of P in order to generate graph clusters with the same size. However, we cannot generate the graph structure very well if we evenly partition it. In our work, the value of P is depended on a graph type, and we have no constrained value. Thus, the actual performance is highly impacted by a graph type, and we discuss this details in Section 5.2.

2.3.2 Multi-Level Approach

In addition to the expensive force computation, another drawback of force-directed layout algorithms is the sub-optimal result of a large graph. Different multi-level approaches have been proposed to overcome this. Gereon et al. [6] surveyed graph clustering algorithms (e.g.,

edge collapse, independent set merger, solar merger, etc.). Frishman et al. [20] proposed a spectral based graph partitioning algorithm, and Muelder et al. [50] discussed a modularity based hierarchy cluster algorithm. We have summarized above methods, and propose our approximated force-directed to weave into the multi-level paradigm for big graph layouts.

In this work, we adopt the solar merger to build multi-level graphs. The solar merger is introduced by Hachul et al. [26] in their fast multiple multi-level methods (FM^3), where each sub-graph is simulated as a solar system. Each node of a sub-graph is classified as sun, planet or moon. The solar system collapses a sub-graph into the sun node of the next level graph. Since a sun node is always the centers of a sub-graph, so it can represent all nodes within its sub-group for repulsive force computation.

In the placement stage, we keep the positions of all parent nodes and initialize their child nodes along a circle, the center of which is their parent node. This design is based on the solar system where child nodes are either earth nodes or moon nodes. After this, we apply our approximated force-directed layout method by traversing graphs from the top level to the bottom one. In summary, the three steps of our multi-level graph layout algorithm are as follows:

1. ***Coarsening***: we use the solar merger [26] as our graph coarsening algorithm to generate a sequence of graphs G^1, G^2, ..., $G^{coarsest}$, where the maximum number of nodes in the coarsest is 50.

2. ***Placement***: in order to initialize the layout for the next level graph, we keep a sun node position, and assign its child nodes along a circle with the center of their parent node.

3. ***Layout***: we use our approximated force-directed algorithm for each level graph layout. We parallelize our algorithm to achieve real-time interaction for graphs with millions of nodes.

The multi-level scheme is used for obtaining high quality results, but cannot provide high performance for a single iteration when generating a large graph layout. The GPU based

approximated force-directed algorithm is the key to satisfy the demand of interactivity. In other words, other graph layout algorithms can be applied to obtain the initial graph layout, and then our GPU graph layout algorithm is deployed for updating graph layout on the fly.

2.4 GPU Implementation

This section describes how the GPU is utilized to accelerate the approximated force-directed layout algorithm. We begin with the discussion of a GPU friendly data structure to represent the multi-level graphs, and then we discuss our GPU parallel force-directed algorithm design.

2.4.1 Data Storage

There are two types of data structures to represent a graph: adjacency matrix and adjacency list. Compared with the adjacency list, the adjacency matrix suffers scalability issues for large graphs. Thus, we use a modified adjacency list, which is similar to the compressed sparse row (CSR) format used by Godiyal et al. [23]. Then we extend the modified adjacency list to organize multi-level graphs.

Figure 2.2 describes the GPU data structure for a three-level graph. Except the finest level graph, there are two groups of data arrays to describe graph clustering and edge connections in each level. The first group includes: *cluster-id*, *cluster-parentIndex*, *cluster-weight*, *cluster-size*, *cluster-offset*, which are used to describe clustering organization of graph nodes. *cluster-id* stores the nodes of the current level graph, which are the sun nodes of the lower level graph; *cluster-parentIndex* provides the indexes of the sun nodes for the next level graph; *cluster-weight* presents the number of nodes of the finest level graph in current cluster; *cluster-size* describes the number of nodes collapsed into this node from its lower level graph; and *cluster-offset* is the actual index of nodes in a sub-graph. For example, from graph G_1, we know that the node ID_0 has its parent node ID_3 (based on *cluster-parentIndex* and *cluster-id* of G_2), and it is collapsed by 3 nodes (*cluster-size*) of its lower level graph, the index of this cluster is 0 (*cluster-offset*), then we know this cluster has nodes ID_1, ID_2, ID_0 from *cluster-id* of

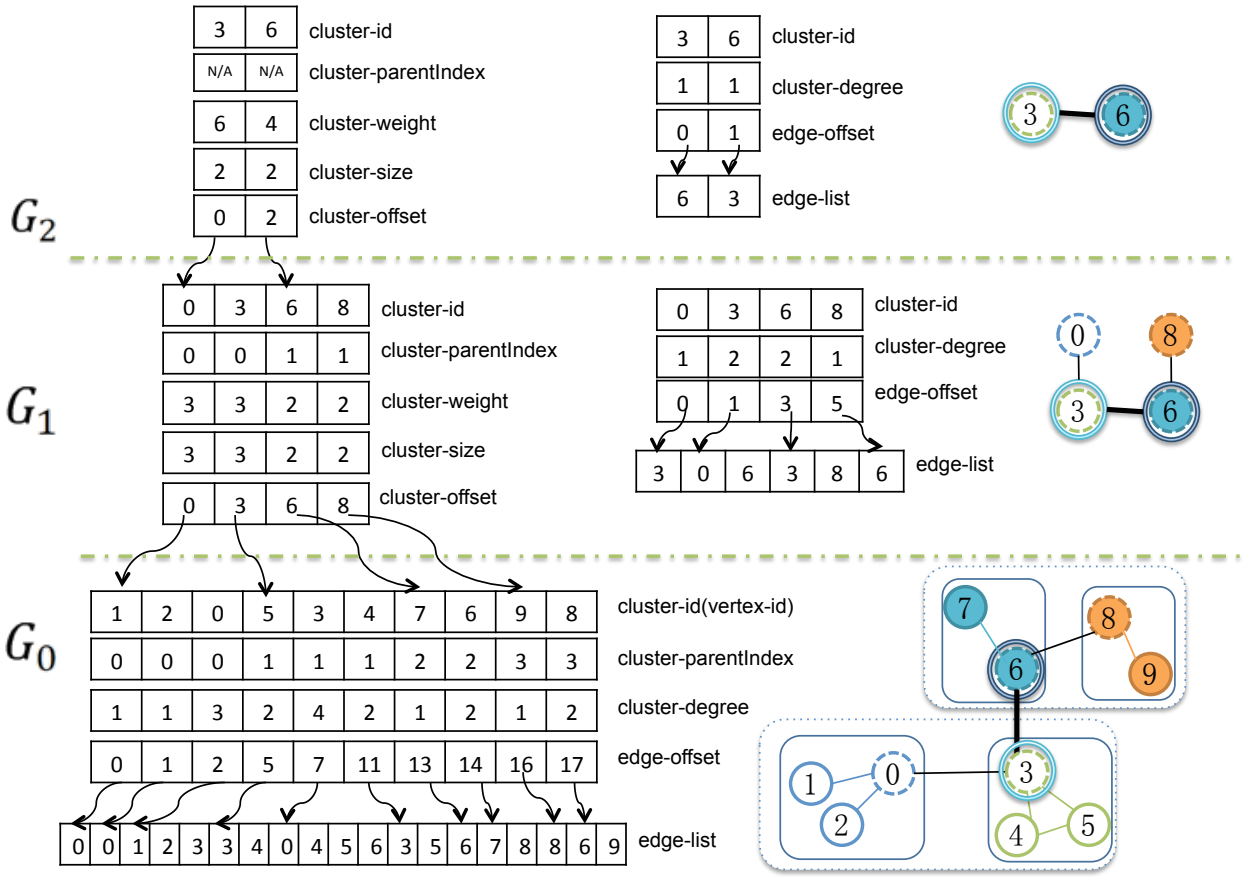

Figure 2.2: An example of the GPU memory organization of three-levels of graphs. Node-link diagrams (right) show the actual level graph. Several data arrays are stored in GPU memory to represent these graphs. In this figure, G_0 is the original graph, which is partitioned into four sub-graphs. G_1 is the graph coarsen by the original graph, which is partitioned into two sub-graphs. G_2 is the coarsest graph.

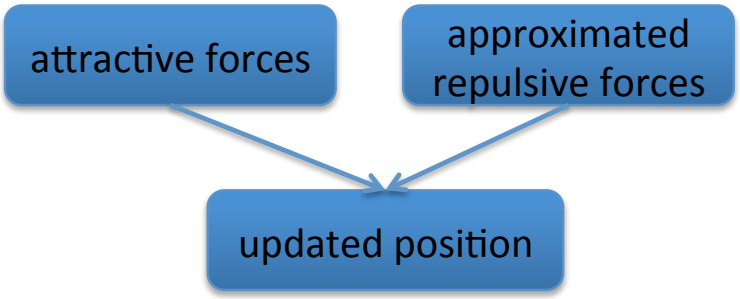

Figure 2.3: The GPU computation flowchart for a single level graph layout. The kernels of *attractive forces* and *approximated repulsive forces* are described in Algorithm 1 and Algorithm 2.

graph G_0. The second group represents the edge connections of each level graph: *cluster-degree*, *edge-offset* and *edge-list*. *cluster-degree* tells the number of edges that other clusters connect to the current cluster; *edge-offset* stores the beginning index of the adjacency edge list; *edge-list* shows the adjacency edge list of current level graph. For example, in graph G_1, we find that node ID_6 is connected with two another two nodes from *cluster-degree*. The beginning index of the adjacency list is 3 from *edge-list*, so we can find the adjacent nodes are ID_3, ID_8 from *edge-list* array.

In addition to the mentioned data arrays, we need other data structures during force calculation. $Vert_{pos}$ holds the positions of all the nodes in the graph. F_{att}, F_{rep} and $Vert_{displacement}$ respectively store the attractive forces, repulsive forces and node displacements of each layout iteration. In summary, the total memory usage is linear with the total number of nodes and edges (the memory usage is constrained by a original graph instead of the value of P). Currently, the GPU memory has already reached to several Gigabytes, which guarantees that we can handle a large graph with millions of nodes efficiently.

2.4.2 GPU Kernels

Our parallel algorithm includes three GPU kernels, shown in Figure 2.3. Each thread in the *attractive forces* kernel and *updated position* kernel is responsible for one node of the selected level graph, which calculates its attractive force and updates its position based on its total force.

Algorithm 1: Attractive Force Kernel

Input : *level*, user-chosen graph level
cluster-id, an array of node IDs
cluster-degree, an array of node degrees
edge-offset, an array of node offsets
edge-list, an array of edge list
Output: F_{att}, an array of attractive forces

```
1 for i = 0 to size(cluster-id[level]) − 1 do in parallel
2     id = cluster-id_level [i];
3     degree = cluster-degree_level [i];
4     start = edge-offset_level [i];
5     for j from 0 to degree-1 do
6         adjacent-id = edge-list_level [start + j];
7         F_att(id) += AttForce(id, adjacent-id);
8     end
9 end
```

Algorithm 1 shows the attractive force computation, where the data arrays about edge information are needed. The kernel of *approximated repulsive forces* is more complex, which calculates the approximated repulsive force based on the data arrays with graph clustering information. Algorithm 2 shows the details about it: the while loop in Line 2 describes the top-down graph traversing for the computation of approximate repulsive force. In each level graph, there are two parts: computing the *internal-electric-force* and passing the repulsive forces as *external-electric-force* to the lower level graph. The first part includes Lines 4-23, with Lines 4-9 describing the special case of the top level graph layout. The second part is about expanding the repulsive forces from one level graph into a lower level (Lines 24-35).

Algorithm 2: Approximated Repulsive Forces

Input : *level*, user-chosen graph level
cluster-id, an array of nodes IDs
cluster-weight, an array of node cluster weights
cluster-parentIndex, an array of node parent indexes
cluster-offset, an array of node cluster offsets
cluster-size, an array of node cluster sizes

Output: F_{rep}, an array of repulsive forces

```
1  curLevel = TotalLevel −1
2  while curLevel ≥ level do
3      size = size( cluster-id_curLevel) − 1
4      if curLevel ≡ TotalLevel −1 then //the top level graph
5          for i = 0 to size do in parallel
6              neighb = cluster-id_curLevel [j]
7              w = cluster-weight_curLevel [j]
8              F_rep(id) += RepForce(id, neighb) ∗ w
9          end
10     else
11         for i = 0 to size do in parallel //repulsive forces within graph cluster
12             id = cluster-id_curLevel [i]
13             index = cluster-parentIndex_curLevel [i]
14             start = cluster-offset_curLevel+1 [index]
15             size = cluster-size_curLevel+1 [index]
16             end = start+size
17             for j = start to end do
18                 neigh = cluster-id_curLevel [j]
19                 w = cluster-weight_curLevel [j]
20                 F_rep(id) += RepForce (id, neigh) ∗w
21             end
22         end
23     end
24     if curLevel > 0 then //pass forces to next level
25         for i = 0 to size do in parallel
26             id = cluster-id_curLevel [i]
27             start = cluster-offset_curLevel [i]
28             size = cluster-size_curLevel [i]
29             end = start+size
30             for j = start to end do
31                 childrenID = cluster-id_curLevel−1 [j]
32                 F_rep(childrenID) = F_rep(id)
33             end
34         end
35     end
36     curLevel -= 1
37 end
```

Table 2.1: A summary of the five tested graphs.

Graph	Number of Vertices	Number of Edges
crack	10,240	30,380
finan512	74,752	261,120
web-Stanford	255,265	1,941,926
grid-mesh	1,000,000	1,998,000
roadNet-TX	1,379,917	3,843,320

2.5 Results and Discussion

We tested our algorithm on a desktop computer running Windows 7 Enterprise, which is equipped with an Intel i7 processor and an NVIDIA GeForce GTX 680 graphics card programmed with CUDA 7.5. We implemented our proposed algorithm in a CPU version and a GPU version. In addition, we choose FM^3 [26], a classic force-directed layout algorithm widely used for large graph layout [20, 27, 60], as a baseline to evaluate the performance and visual result.

In total, we picked five different graphs as five tested cases for our algorithm, including one artificial graph generated by ourselves and other four well-known graphs coming from the University of Florida Sparse Matrix Collection[1], Chris Walshaw's graph collection[2] and Stanford Large Network Dataset Collection[3]. Table 2.1 shows a brief summary of these five graphs.

2.5.1 Visual Assessment

Making a fair, reasonable and comprehensive evaluation of visual representations (e.g., layout) of a graph is difficult because the assessment may be highly subjective (e.g., when considering the aesthetics aspect). Some previous large graph layout algorithms attempt to generate visually pleasing graph layouts but ignore the capability to enable human interactions to adjust layout results. Alternatively, we emphasize the importance of improving visually pleasing layouts with interactive capabilities. This can potentially help to generate

[1]http://www.cise.ufl.edu/research/sparse/matrices/
[2]http://staffweb.cms.gre.ac.uk/ wc06/partition/
[3]http://snap.stanford.edu/data/

more meaningful layouts for a large graph, especially from a human perceptible perspective.

Figures 2.4 and 2.5 show the results of our multi-level algorithm (200 iterations per level without human interactions) and FM^3 for graph *crack* and *finan512*. For both graphs with evenly distributed nodes (Figure 2.4) and those with unevenly distributed nodes (Figure 2.5), results from our method can achieve at least the same visual quality as those from FM^3 (e.g., C v.s. D in Figure 2.4).

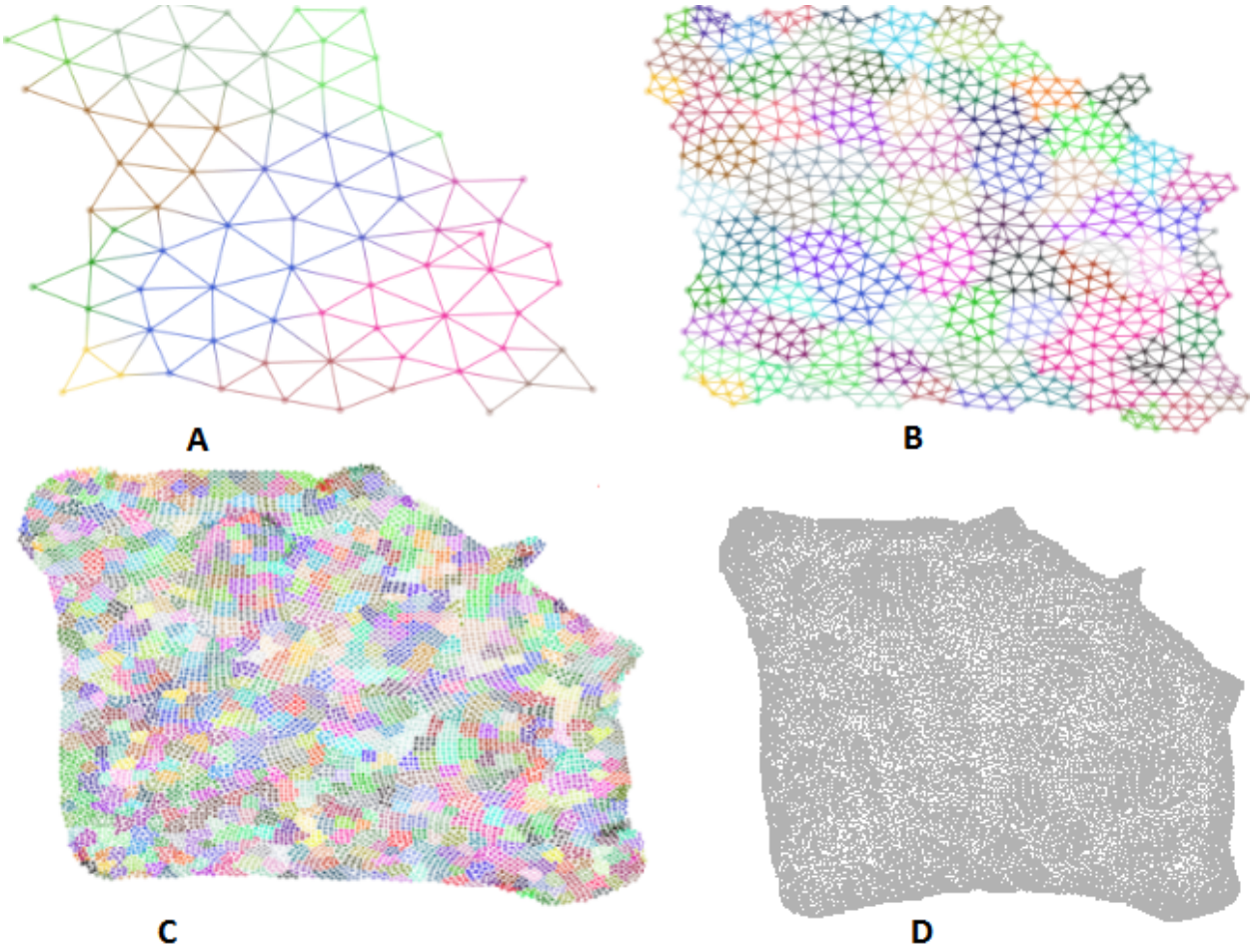

Figure 2.4: Results for graph *crack*. A, B, C are the layout results of three level coarsen graphs based on our algorithm (graph nodes from a same cluster are in the same color). D is the result from FM^3 implemented by ODGF [11].

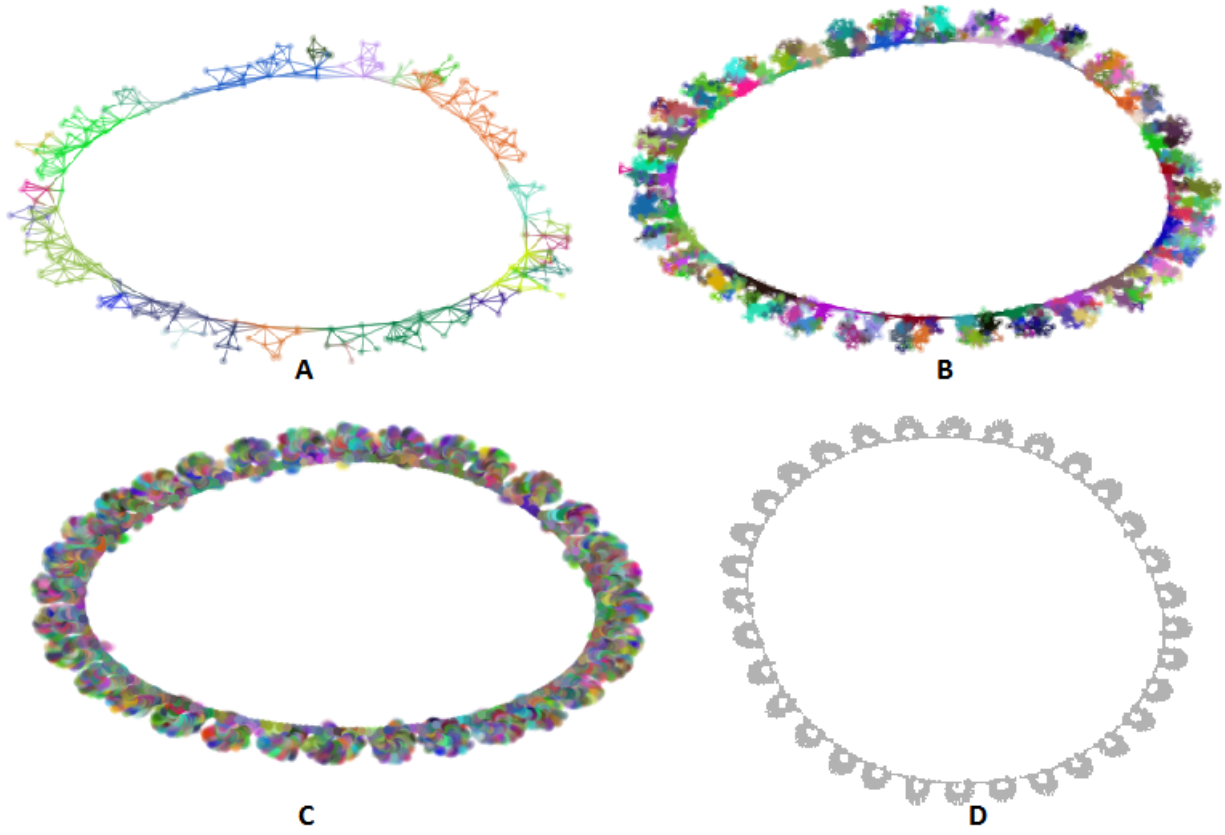

Figure 2.5: Results for graph *finan512*. A, B, C are the layout results of three level coarsen graphs based on our algorithm (graph nodes from a same cluster are in the same color). D is the result from FM^3 implemented by ODGF [11].

2.5.2 Performance Analysis

Termination conditions of force computation impact the quality and performance of graph layouts. Previous works (e.g., [20], [23]) focus on generating static pleasing layouts of big graphs, and they select a fixed number of iterations to halt the force computation. However, simply using a fixed number of iterations as a termination condition cannot guarantee that the computed layouts always satisfies user requirements, especially considering the fact that different users may focus on different aspects (e.g., aesthetics) to evaluate graph layouts. Moreover, Endert et al. [15] emphasized that human is the key for exploratory knowledge discovery and argued that "human is the loop" for sensemaking. Thus, *it is important to* ***support user interactions for graph layouts*** to make sense of large graphs.

The support for interaction with large graph layouts is a key design concern in our work, so the average computation time for one iteration, rather than the total amount of time for getting the final layouts, is used as a meaningful measurement for us to evaluate the performance of our algorithm. This one-iteration oriented computation time is calculated by averaging the time of all iterations needed in our algorithm to achieve the finest level graph. We run both our algorithm and FM^3 (implemented by ODGF [11]) at least 20 times to collect enough data (statistically meaningful) for comparison. In addition, we consider rendering time because it directly impacts the actual visual layouts presented to users.

Table 2.2 summarizes the measured performance metrics of each tested graph. Based on the first two rows, it is clear that the CPU version of our algorithm, on average, runs at least 7 times as fast as FM^3 does. For graphs with millions of nodes, the performance of our algorithm remains relatively stable, compared to that of FM^3. The worst performance of our algorithm (GPU Version) is much smaller than 500ms [44], which verifies that the performance of our algorithm can support real-time user interactions for graphs with millions of nodes.

By comparing columns 4 and 5, we find that the performance of our algorithm (GPU version) for mesh-like graphs is better than that for the small-world graphs. The reason for this is that we can get evenly partitioned graph clusters based on the solar merger for mesh-like graphs. However, the node degree of small world graphs is unevenly distributed, following the power law distribution. As a result, the distribution of sub-graphs also follows the power law distribution, when applying graph clustering algorithms (e.g., Figure 2.6 shows the sub-graph distribution of *web-Stanford*). Thus, the GPU threads for *internal-electric-force* computation suffer the workload imbalance problem. To handle this problem, a graph coarsening or partition algorithm that evenly partitions a graph is a better choice (e.g., Frishman et al. [20] spectral graph partition algorithm). However, evenly partitioned graphs may not obtain the optimal graph topology structure well (spectral based algorithms cannot work in our case), especially for small world graphs. This makes it more difficult for the force-directed layout algorithm to converge into a global minimal configuration. To balance the trade-off between performance and quality of layouts, we suggest customizing the graph

Table 2.2: Performance results (milliseconds per iteration) on each tested graph. The rendering time only includes graph nodes rendering.

Performance	crack	finan512	web-Stanford	grid-mesh	roadNet-TX
FM^3 on CPU	63.900	630.000	2581.399	7814.100	11484.799
Our approximated force calculation on CPU	6.799	83.738	337.049	476.548	473.564
GPU Kernel Attractive-Force	0.315	0.660	38.177	1.287	1.767
GPU Kernel Approximated Repulsive-Force	2.099	3.887	28.184	14.484	17.819
GPU Kernel Updated-Position	0.007	0.008	0.015	0.013	0.011
GPU Others	0.302	0.383	0.728	1.296	1.727
GPU Total	2.732	4.938	67.104	17.080	21.324
Rendering	0.809	1.288	2.252	4.958	9.727

partition algorithms. For example, we can give a threshold of maximum cluster size P to avoid generating giant clusters.

Another possible solution is using the stochastic force-directed layout algorithm discussed in [10]. For giant clusters, the size of which are larger than the threshold, we randomly pick neighbor nodes from the cluster for the *internal-electric-force* computation.

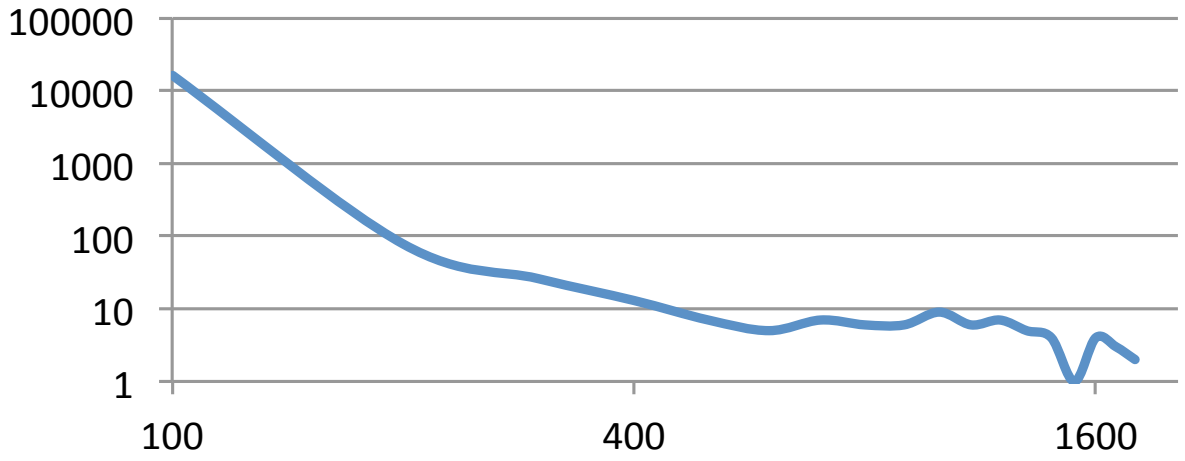

Figure 2.6: The sub-graph distributions of *web-Stanford*. The X-axis is the number of vertices, the Y-axis is the number of sub graphs. It is a power law distribution.

We have tested the scalability of our algorithm for mesh-like graphs. We generate successive grid-meshes, which can be evenly partitioned by the solar merger. Thus, the repulsive force computation of mesh-like graphs avoids the workload imbalance problem. Figure 2.7 shows the performance of our algorithm (GPU version) for these grid-meshes. It is clear that our algorithm can scale to mesh graphs with millions of nodes, and also guarantees real-time user interactions.

It is hard to compare our GPU version with other GPU implementations. Apeksha et al. [23] provided the total time to generate appealing graph layouts without giving the number of iterations. Besides the GPU based force computation, their algorithm spends extra time on tree structure building (CPU) and CPU-GPU data movement. Frishman's [20] reported their algorithm is 2-4 times faster than FM^3, and their GPU implementation can achieve 5.5 times faster than their CPU version. However, their algorithm takes more time for each iteration

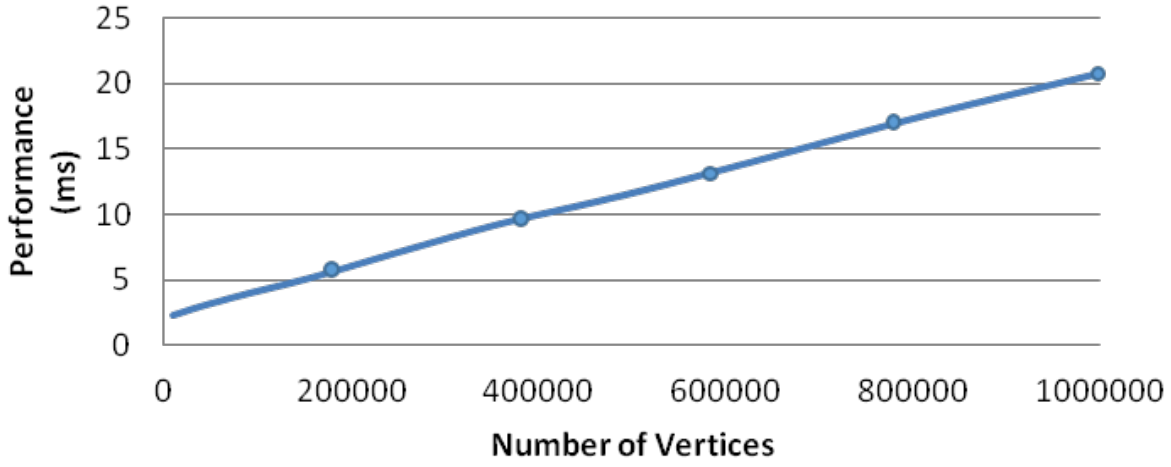

Figure 2.7: The performance of grid-mesh layouts. The X-axis is the number of vertices, and the Y-axis is the performance. The averaged P of each cluster is 9 based on the solar merger algorithm.

computation, where each level sub-graphs' position needs to be updated. Even though our algorithm needs more iteration numbers to achieve similar results (200 iterations per level compared with Frishman's 50 iterations per level), we can achieve real-time frame-rate for large graph layouts.

We use the OpenGL VBOs (Vertex Buffer Objects) to efficiently render graphs. This avoids data movement between the CPU and the GPU. The last row of Table 2.2 shows the rendering performance: visualization of a million nodes only takes about 5 milliseconds. Based on our GPU implementation, we can directly map the GPU memory to OpenGL VBOs without any CPU-GPU memory communication. Moreover, the multi-level graphs are stored and updated by the GPU, without any communication with the main memory. This implementation keeps all graph data in the GPU memory, which fully utilizes the GPU resources for efficiently computing and rendering.

2.5.3 Discussion

The evaluation of a graph layout algorithm is challenging. Stress model based force-directed algorithms (e.g., [38]) use the stress error function to quantify layout quality and terminate the computation [36]. These algorithms need to store pairwise nodes' distances, so they do not scale well for graphs with millions of nodes. Our method (spring-electric model) is based on adjacency lists, which can be applied to big graphs. However, termination conditions (e.g., the energy threshold and local minimum) of spring-electric model still need further investigation. Our work attempts to enable users to terminate the computation and modify the layout results, so we focus on a single iteration of force calculation in the performance analysis.

One limitation of our algorithm is that it only considers the scalability of the repulsive force computation. For complex graphs, the number of edges may be linear with squared number of nodes. In this case, computing attractive force becomes a bottleneck. Compared with other cases, our algorithm may not handle this well.

2.6 Usage Scenario

In this section, we present two case studies to demonstrate how users interact with a big graph for sensemaking activities. We first demonstrate that a user can explore a big graph structure based on the multi-level paradigm. Then we show that a user can interact with graph layout to perceive more semantic meanings.

2.6.1 Visual Exploration of Web Networks

Suppose that Mike is a network analyst and he attempts to analyze the structure of the *web-Stanford* graph. He first processes this graph using the solar merger to generate four separated abstract levels, which leads to a total of five level graphs. Then he explores these graphs in top-down order. When he is satisfied with the layout result of one level of the

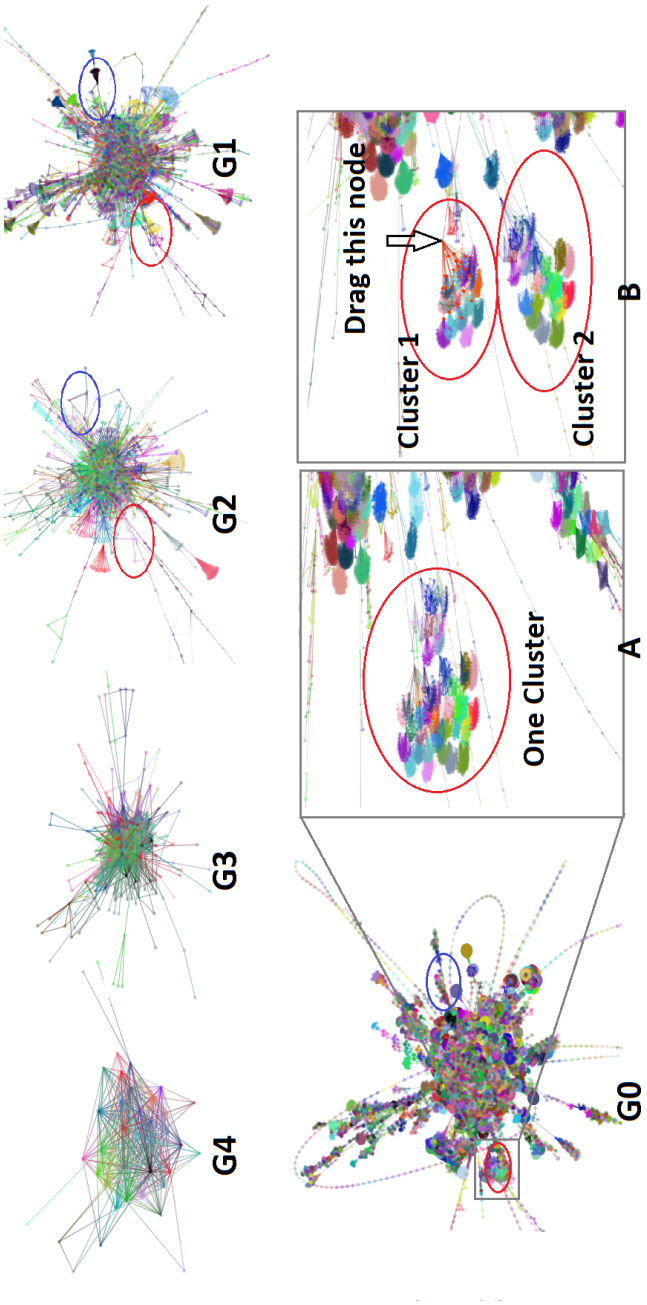

Figure 2.8: Five level graphs layout of *web-Stanford* data (graph nodes from a same cluster are in the same color). From top to bottom, those graphs are G_4, G_3,G_2,G_1,G_0. The red and blue circle are the highlighted sub-graphs in our case study. Picture A shows graph detailed layout when a user zooms into the red circle. Picture B shows that one cluster separates into two clusters when a user drags a hub node.

graph, he terminates the computation, adds nodes and edges and then explores the next level layout. Figure 2.8 shows different level of graph layout results in this process.

By comparing these different levels of graph layout results, Mike notices that two sub-graphs show different evolutions. These two parts are circled in red and blue respectively in Figure 2.8. The basic structures of these two sub-graphs are in the shape of a triangle. After Mike moves to the next level graph, the sub-graph circled in red turns to a clique-like structure, while another sub-graph that is circled in blue changes to a tree-like structure. In the finest level graph, the former becomes a compact cluster but the latter still remains a tree-like structure. The difference in the structure evolutions of the two sub-graphs catches Mike's attention, especially the first sub-graph. Mike wants to better understand why the structure of the first sub-graph has relatively greater changes, so he decides to zoom into the former (circled in red) to see more details for further exploration.

Figure 2.8 (A) shows the detailed graph layout result. This cluster-node consists of multiple small clusters that overlap with each other. Mike finds a hub node of this cluster and drags it a little bit to see what would happen. It turns out that several small cluster-nodes dynamically move following the dragged one shown in Figure 2.8 (B). After this, Mike quickly realizes that this sub-graph can actually be separated into two clusters.

In summary, with our proposed technique, Mike is able to see the evolution of big graph layouts by exploring different level graph layouts. With the capability of dragging nodes and modifying the layout of graphs in real time, the proposed algorithm supports users flexibly exploring large graphs. This potentially combines the human cognition with computation power (e.g., graph layout algorithms) to better support human sensemaking with large graphs.

2.6.2 Visual Exploration of Co-authorship Networks

In this case study, we present how an analyst makes sense of a co-authorship network. Suppose Grace is a social network analyst. She is interested in how research scientists

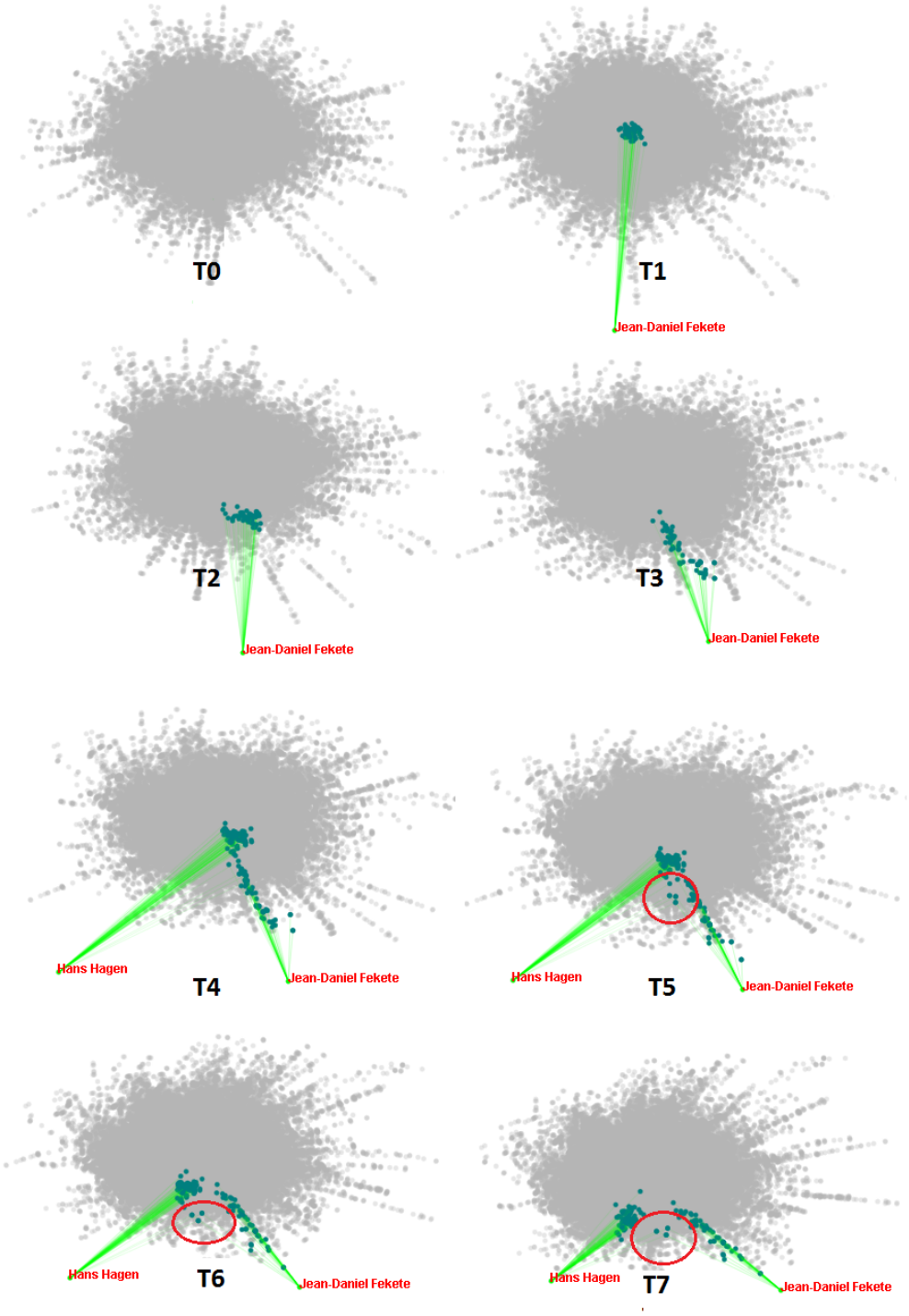

Figure 2.9: The graph layout of the DBLP co-authorship network. The figure shows series of node movements. Figure T0 shows the initial graph layout based on FM^3 algorithm. In figure T1, *Jean-Daniel Fekete* is dragged away from the center of network. T2 and T3 are the snapshots of node movements. In figure T4, *Hans Hagen* is dragged away from the center of network. T5, T6, T7 shows node movements.

collaborating with each other. Thus, she downloaded publications from DBLP[4], parsed them and generated a co-authorship network (515,103 nodes and 1,856,690 edges). At first, she obtained a graph layout based on FM^3 as shown in Figure 2.9 (T0). The visualization is a dense and intermingled network. The "core" of network contains most of nodes and edges, which has a nontrivial structure. The research communities cannot be separated using FM^3. Thus, she changes to use our algorithm and interacts with the graph layout.

First, she identifies research communities by their central actors (hub node of a subgraph). She is interested in a research community with *Jean-Daniel Fekete* (JDF) as a central actor. To separate the JDF community from the "core" network, she drags this central actor away from the "core" in Figure 2.9 (T1). The nodes belonging to this community move towards the central actor. By observing nodes' movement of figures T2 and T3, she easily identifies that nodes are separated into two groups. One group of nodes moves slow, and they "blend into" the network, while the other group of nodes moves fast and they are separated well from the "core". She checks details of each node, and finds that the first group consists of senior researchers, while the other consists of junior researches.

She hypothesizes that some senior researchers may not only belong to one research community, but multiple areas. To verify it, she drags *Hans Hagen* (HH) away from the network center in figure T4. Then she observes node movements, and finds that three senior researchers have collaborations with both of them, as shown in Figure 2.9 (T5, T6 and T7).

Compared with previous approaches, Alice gains more information by interacting with the graph layout using our algorithm. She easily separates research communities from a large nontrivial network for analyzing its structure. She also find researchers across multiple areas.

2.7 Lessons Learned

In this chapter, we propose to enable human interaction with large graph layouts. Based on this key design consideration, we contribute a topology based method to accelerate a force-

[4]http://dblp.uni-trier.de/

directed graph layout algorithm that can support real-time exploration of large graphs. We parallelize this method by using the powerful resources of the GPU. We summarize two lessons learned about designing accelerated algorithms to handle big graphs as follows:

- **Taking advantage of parallelization**. Performance is a key concern for designing algorithms to handle big data (e.g., large graphs). Parallelization is one option to accelerate algorithms. GPU has been praised for its significant performance improvement and programmability for general purpose computation. Therefore, researchers can rethink existing algorithms and utilize the GPU for *parallelization*.

  We design parallel algorithms considering two aspects: *data structures* and *independent instructions*. We use modified adjacency lists to organize multi-level graphs and separate our force-directed algorithm into different kernels to reduce its dependency.

- **Light computation per iteration to enable human interactions**. Figure 2.10 shows the flow chart of our large graph layout system, where force computation is the bottleneck. Previous graph layout algorithms cannot support human interaction since they have heavy computations per iteration. To support sensemaking with large graphs, we emphasize *light computation per iteration*. Our topology based method needs less computation compared to nodes' distribution methods since we save more calculation for approximated force. This enables humans to be involved in the process of graph layouts. By adjusting the automatically generated layouts from computation, this mixed approach for graph layouts can lead to the results that make better sense for individual users.

  Based on the flow chart of our layout system in Figure 2.10, humans are involved into the layout process and they can halt the computation in advance to reduce total time. Actually previous algorithms use a fixed number of iterations for large graph layout to guarantee a converged layout result, which may waste some iterations and increase the total performance. Striving for the best solution to halt layout computation may introduce more memory and computation requirements that may increase total time to layout a large graph. In such case, light computation per iteration may be a good design choice for obtain better layout result in a short time by involving human interaction.

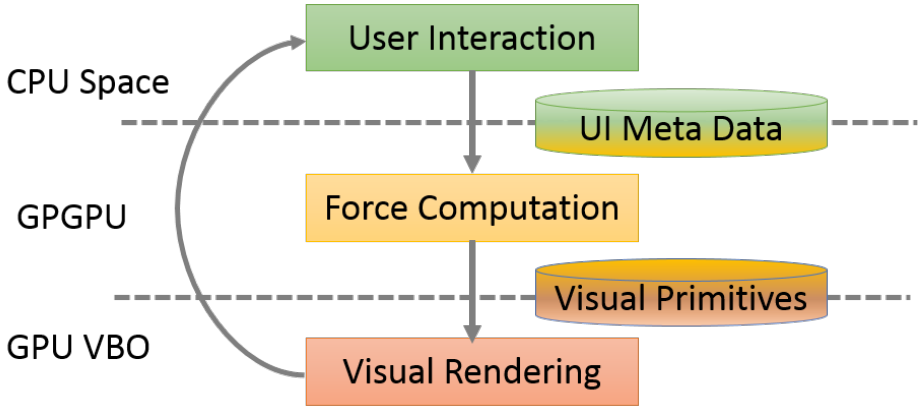

Figure 2.10: The flow chart of our large graph layout system. User interactions are involved for generating graph layouts. User interaction data are passed to the GPU side for force computation. Visual primitives are updated based on force calculation.

However, light force computation may need more iterations to obtain a converged graph layout. This cannot guarantee a decrease in total performance. So the design for large graph layout algorithms involves a trade-off between the total performance and per-iteration performance. We favor light computation that can support human interaction for large graph layout and exploration.

2.8 Summary

To enable human interactions with a large graph layout, we propose a fast force-directed layout algorithm with detailed GPU implementation. Our algorithm highlights two contributions. First, we propose to calculate the approximated repulsive force based on the topology of a graph, instead of the spatial distribution of its nodes, which avoids building and traversing a spatial indexing data structure. Second, we give our GPU implementation, which includes the GPU kernels' design of the force-directed layout algorithm and the GPU memory layout of multi-level graphs.

We demonstrate layouts by testing several well-known graphs. Our method can generate visually pleasing graph layouts for the tested graphs, and provides performance that potentially supports user interactions with large graphs. We also present two case studies to

demonstrate our algorithm can support users in exploring large graphs and refining their layouts on the fly to investigate more details. We summarize lessons learned from this work for designing algorithms to enable real-time human interactions with large graphs for sensemaking. We hope these lessons can improve the future design of algorithms using interactions for sensemaking of large graphs.

Chapter 3

AVIST: A GPU-Centric Design for Visual Exploration of Large Multidimensional Datasets

3.1 Introduction

With the rapid growing of data, interactive visual analysis becomes a key for gaining usable insights from massive data. To support sensemaking of big data, exploratory visual analysis emphasizes three aspects. First, *flexible data filtering* (e.g., cross-filtering [63, 64]) which empowers analysts to "prune" large datasets to gain buried relationships. Second, *fine data details* that should be provided on demand, so that analysts can capture certain subtle abnormal activities. Last, *exploration data analysis* which is usually an interactive and iterative process. Thus high performance is necessary for exploration of big data, which requires real-time frame-rates. For example, when cyber security analysts attempt to identify attacks from millions of records in network traffic logs, they have to filter data records by iteratively using different data attributes to finally narrow down to the suspicious records.

However, there are many challenges to achieving these goals when analyzing big data. When the data becomes bigger, it cannot fit into memory or the computation speed is not sufficient

for real-time performance. In order to feed more data into memory, Tableau [1] highlights the Tableau Data Engine (TDE) [65]. The TDE uses a specialized column-oriented format, which has high data compression ratio (considering the repeating values in the columns) and can provide high level data aggregation information. To achieve fast visual querying of big data, imMens [45] utilizes the GPU's parallel computing to improve performance. First, it aggregates data into data tiles using the data cube technique [24]. Second, it uses WebGL for data processing and rendering for increased performance. Both Tableau and imMens only provide the ability to explore high level aggregation information of big data. However, in order to flexibly investigate fine-grained data details, such as identifying buried data relationships by complex filtering, those methods cannot be directly applied, thus new techniques need to be investigated.

This chapter tackles the problem of visual exploration of big data, and aims to help analysts get the fine-grained details by exploratory visual analysis of big datasets. To achieve this, we propose an Animated VISualization Tool - AVIST, which runs on an off-the-shelf GPU for analyzing millions of multi-dimensional data records per second. We emphasize "*animation*" in the context of time-series multidimensional datasets for identifying data temporal behaviors. The performance of AVIST is judged by FPS (frame per second). In summary, our key contributions in this work are as follows:

1) We propose a GPU-centric design for interactive visual exploration of big datasets, which emphasizes the use of the GPU for data storage and computation.

2) We design a data dependency graph for characterizing GPU based data transformations, which supports data aggregation and visualization on demand.

3) We implement AVIST following the GPU-centric design as a proof-of-concept. AVIST features animation and cross-filtering interactions for slicing big data into small data, and the GPU parallel computing for transforming raw data into visual primitives.

4) We present two usage scenarios to demonstrate that AVIST can help analysts identifying abnormal behaviors for large network flows and inferring new hypotheses for international trade transactions.

3.2 Related Work

To achieve visual exploration of big datasets, four questions outline the discussion of related work: 1) which type of data should be visualized, and what are its characteristics; 2) how does one store and manage big data; 3) how does one compute big data and achieve fast performance; 4) how does one explore and interact with big data.

3.2.1 The Exploration of Multidimensional Datasets

Large multidimensional datasets are very common and ubiquitous. Most of them usually consist of behavioral data that are generated by people or machines, which have a natural temporal ordering (streaming data) [8]. Besides the large volume, high velocity is also an important consideration for analyzing such big datasets [9, 70]. For example, network logs and financial transactions can be generated millions per second. When faced with large volume and high velocity multidimensional datasets, analysts may not know what they are looking for; they may know something is interesting only after they find it. In this setting, analysts have no idea how to formulate their queries. Thus, the data exploration of large time-series and multidimensional datasets is a key ingredient for knowledge discovery [35].

The exploration driven system for knowledge discovery becomes a novel requirement in the era of "big data". The data exploration becomes a first challenge for making sense of large streaming multidimensional datasets. Our work tackles this problem and provides a solution for exploring the large volume and high velocity datasets.

3.2.2 Big Data Management

Big data management methods vary based on the size of datasets. If a dataset can fit into a computer memory, in-memory databases should be investigated. When the dataset size is larger than the computer memory capacity, data prefetching and out-of-core techniques need to be considered [54]. If a dataset is even bigger than the computer disk capacity, the distributed file systems (e.g., Hadoop Distributed File System [56]) should be considered.

Our work aims to achieve real-time performance for visual exploration of big data on an off-the-shelf desktop. Thus it only considers datasets that can fit into computer memory. Previously researched tools (e.g., imMens [45]) use one million or more data items as a threshold. Therefore, we follow this definition and give an upper bound of big data: the capacity of the computer memory.

There are two general ways to manage the multidimensional (tabular) datasets: row oriented and column oriented databases. Abadi et al. [2] discussed the differences between them, and pointed out that the column format can apply compression techniques to significantly reduce data size and improve query performance. For example, Tableau's TDE is a column oriented format for managing big data. The row oriented database is better for analyzing small transactions to "find a needle in haystack". Thus, both of them are widely used in database community.

Data modeling, sampling and aggregation [45] are other ways for handling big data. These methods reduce big data into smaller data based on data precomputation. However, these kinds of big data visual explorations are constrained by precomputation methods, and cannot provide flexibility and low level data details. Our work emphasizes visual exploration of data fine-grained details using complex filtering and highlighting. Thus, we manage big multidimensional datasets in a row oriented format, and utilize parallel computing for visual exploration of big data on demand.

3.2.3 The GPU Acceleration

Contrasted with the CPU, graphics processing unit (GPU) has two unique features.

- More cores and fine levels of parallelism. The GPU has a many-core architecture, which may include thousands of cores. This feature makes the GPU specialized at compute-intensive, highly parallel computation.
- Higher memory bandwidth. This means that the GPU can access data at higher speed (usually more than 100GB/s), or more data in a fixed time period, when compared to

the CPU.

The GPU is more popular in scientific computation and visualization [4], and several factors have hampered its use for general-purpose computation in information visualization and visual analysis fields. Primarily the GPU lacks complex cache organizations and sophisticated branch prediction, so complicated sequential algorithms are hard to parallelize on a GPU platform. However, the development of GPU libraries (e.g., Thrust[1] and cuBLAS[2], etc.) help translate these algorithms on the GPU. Another consideration is slow transfers from CPU to GPU memory, and the limited GPU memory size. However, transfer speeds have improved significantly thanks to PCI-bus improvement, and now the GPU memory capacity is quite good (the latest Nvidia Geforce GTX TITAN Z has 12GB memory). All these GPU developments make it a possible solution for handling big data to support exploratory data analysis [52].

In fact, the GPU has a deep root in graphics and visualization, and it is good at handling millions of vertices. This chapter aims to utilize GPU fine-grained computation parallelism and high memory bandwidth to support exploratory visual analysis of large multidimensional datasets. Thus, we propose AVIST, which is implemented based on a GPU-centric design for visually exploring large time-series and multidimensional datasets.

3.2.4 Big Data Exploration

Querying is one of the fundamental interaction techniques for exploring data. Polaris [57] features a visual querying language by integrating analysis and visualization of multidimensional datasets. Analysts can construct sophisticated visualizations by simple drag-and-drop operations. However, Polaris is incapable of managing big data. Tableau, the successor of Polaris, makes big progress for big data visualization, whereas it lacks flexible visual filtering for investigating data details.

Animation is a type of time multiplexing technique [18] for visualizing large data items.

[1] https://code.google.com/p/thrust/
[2] https://developer.nvidia.com/cublas

Moreover, it can effectively reveal temporal patterns by significantly improving graphical perceptions [32]. Our work highlights animation interactions for slicing big data into fine details. Cross-filtering [64] is a method for flexible visual drilling-down into fine-grained relationships for multidimensional datasets. Hence we emphasize cross-filtering and utilize the GPU to improve its performance, and achieve fast, flexible and detailed data exploration.

3.3 The GPU-Centric Design

3.3.1 Design Principle

The design of AVIST follows two important principles: 1) *fully utilizing GPU fine-grained parallelism and high memory bandwidth to boost performance on an off-the-shelf desktop computer*; 2) *affording flexible visual querying and filtering (e.g., animation and cross-filtering) to visually explore fine level data details*. Based on these two guidelines, we carefully consider the data storage and computation design on the GPU:

- **Data storage**: We store raw data in the GPU memory. Moreover, all derived data is also stored in the GPU memory to take full advantage of GPU high memory bandwidth.
- **Data computation**: We highlight cross-filtering for slicing data records from different attributes (animation for time domain). A data dependency graph (directed acyclic graph) characterizes such data transformations on the GPU to support data aggregation and visualization on demand.

3.3.2 The GPU-Centric Design

There are two key ideas about the GPU-centric design: 1) raw data is stored on the GPU; and 2) data processing is parallelized on the GPU.

We emphasize that the raw data is important for exploratory visual analysis. Because it can potentially help analysts analyze each data record without any information loss (*finding*

a needle in a haystack). The raw data is stored in the GPU memory, which enables: 1) fast data accessing based on its high bandwidth; and 2) avoiding data transfers between CPU and GPU. Besides raw data, derived data is also stored in the GPU memory (visual primitives are stored in the GPU vertex buffer objects).

We also highlight that data aggregation and visualization are based on the GPU parallel computation. The benefits include: 1) vectored computation for utilizing the GPU fine level parallelism; 2) data aggregations and visualizations are on demand rather than precomputed values.

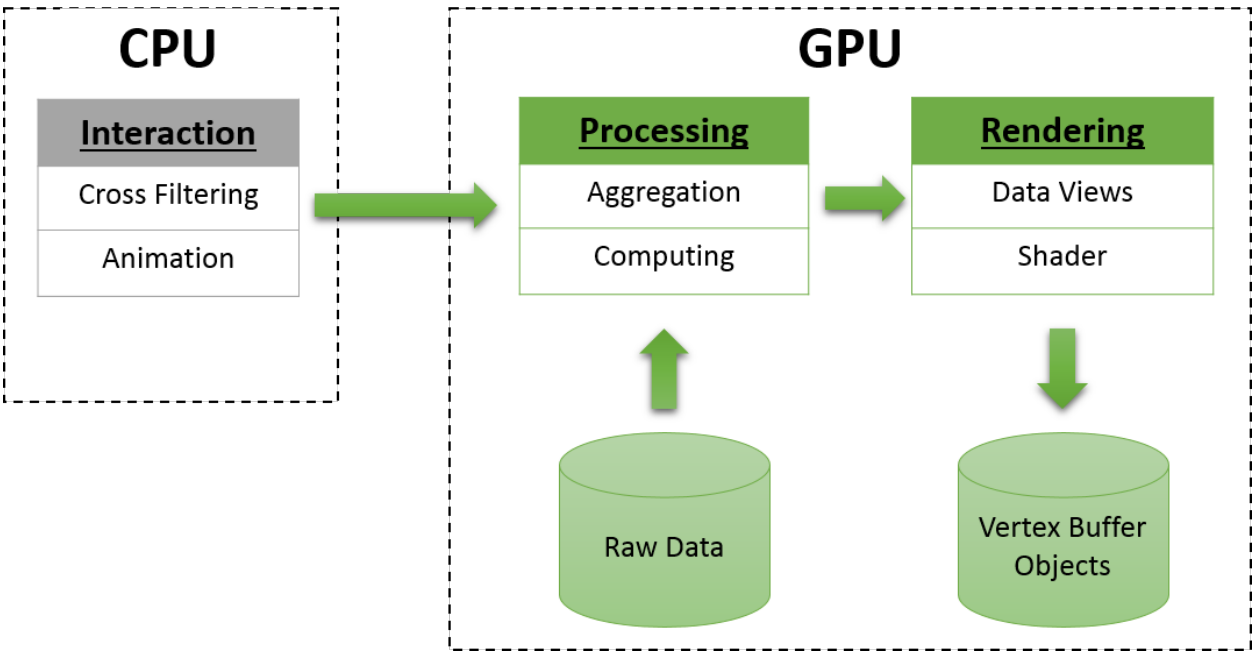

Figure 3.1: The GPU-centric design: user interactions are transfered from CPU to GPU. Data storage and computation are on the GPU.

Figure 3.1 shows the GPU-centric design. User interactions are generated on the CPU side, then they are transferred to the GPU, which triggers the GPU parallel processing and rendering. The GPU processing is also separated into two parts: 1) general data aggregation and filtering; 2) computation of visual primitive in each data view. In all, the GPU-centric design emphasizes that both data storage and computation are handled by the GPU, to fully utilize its resources.

3.3.3 Data Storage

As previous mentioned, the GPU memory capacity limits our GPU-centric design for handling large datasets. To feed more data into the GPU memory, we apply a lossless compression technique for preprocessing multidimensional datasets.

Consider a multi-dimensional dataset, which may include different types of data. First, we identify each type of data, and separate it into time data, quantitative data, and categorical-ordinal data. Second, we use different methods for compressing the data based on its characteristics. For the categorical-ordinal data, we count all possible values and map each value into a unique ID, this key-value map is stored in main memory as meta data. For time and quantitative data, we calculate its minimum and maximum values as meta data.

On the GPU side, we organize raw data in a row-oriented format, where all data records are ordered based on their time values (we emphasize the data temporal and spatial locality for fast data retrieving). We store the binary code of each data item instead of its ASCII code to save memory space. Thus, each *time* data occupies 8 bytes and each quantitative data needs 4 bytes (one *Int* or *Float*). We store categorical-ordinal data IDs instead of their values. The memory requirement varies based on the possible values (one byte is required if a number's IDs is less than 256; two bytes are considered if its ID is less than 65536; other cases use four bytes). Table 3.1 summarizes the compression methods for preprocessing datasets.

Table 3.1: Data preprocessing.

DataType	**GPU memory binary format**	**Main memory meta-data**
Time	time_t 8 bytes	minimum value maximum value
Quantitative	Int or Float 4 bytes	minimum value maximum value
Categorial-Ordinal	$1 \sim 4$ bytes	Dictionary (IDs and data values)

Compared with the row-oriented format, column-oriented data organization has better performance on some analytical workloads. However, we use row-oriented format instead of column-oriented after considering several factors: 1) exploratory analysis cares more about

fine-grained record details rather than high level aggregations, and we want to guide analysts to each data record based on their querying and filtering, rather than show them high level aggregations; 2) column-oriented format primarily works on columns, which are treated individually. Thus, queries for each column work efficiently, while cross-column queries need to retrieve multiple columns and to assemble that data in a complex way. The computation may be very time consuming and cannot be easily vectorized; 3) row-oriented data has a spatial locality based on its IDs. If the data has time, temporal and spatial locality can be unified together for fast data retrieval; 4) the computation of row-oriented data can be easily vectorized by leveraging GPU fine parallelism to improve performance.

3.3.4 Data Computation

To organize GPU parallel computing, a data dependency graph is proposed to characterize data transformations on the GPU. Figure 3.2 shows the detailed data flow design, which is a directed acyclic graph. Here, rectangles represent data, while ovals are parallel computations. The graph is separated into three parts:

- **Data filtering**. Data filters (e.g., *time window*, *filters* and *highlights*) are generated by user interactions on the CPU side, then they are passed to the GPU side. The GPU pipeline of data filtering is as follows. First, the time window is applied to slice raw data into data snapshots. Second, filters are applied to data snapshots by removing uninteresting data records. Last, highlights aim to emphasize important data items from the filtered data list.

- **Data processing**. This step aims to transform the filtered data records into visual primitives. However, data transformation methods vary considering different data views. We summarize them and characterize data processing into two stages for each data view: 1) aggregation for generating geometry data (e.g., binning data for histogram view); 2) computation for transforming geometry data into visual primitives (e.g., the bar height for histogram view).

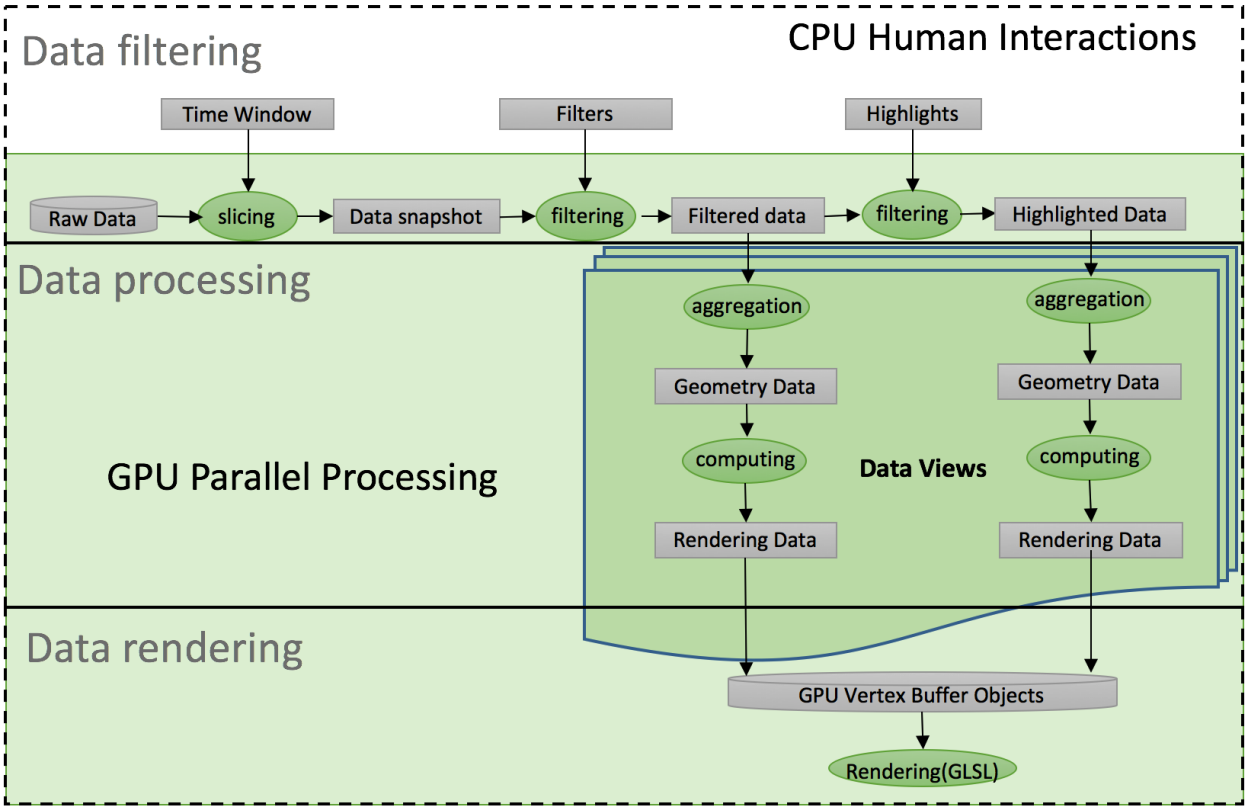

Figure 3.2: The data dependency graph: rectangles represent data and ovals are parallel computations. Data filters are generated on the CPU side and transfered to the GPU. All data and computations are done on the GPU.

- **Data rendering**. All generated visual primitives are stored in the GPU vertex buffer objects. We use the OpenGL shading language (GLSL) to generate the ultimate visual results when rendering visual primitives on the screen (e.g., splitting one vertex into four for rendering rectangles by geometry shader, filling colors in triangles by fragment shader).

The data dependency graph has several advantages. First, it follows the cross-filtering design pattern [63], where the cross-filtering and coordinated multiple views are coupled together. Second, all data processing and visual rendering can be easily vectorized, and data computations are processed on the fly. Third, incremental computation can be applied to exploit temporal and spatial locality. When users play animation, frame-to-frame coherence can be exploited. In such cases, only incremental data should be considered. Moreover, user

interactions are incremental and interactive. So previous querying results can be reused by next queries. Finally, the data dependency graph is flexible and extensible. More data filters and data views can be easily extended.

3.4 AVIST

We implemented AVIST following the GPU-centric design. AVIST is written in C++, its computation codes are based on CUDA 7.5 and visualization codes are based on OpenGL 4.5 and GLSL 4.5. The interface is coded by wxWidgets on the CPU side. We use the Thrust library[3] for accelerating sorting, scaning, and reduction operations.

In this section, we give the implementation details of AVIST from four aspects: 1) the code organization, which follows the Model-View-Controller (MVC) architecture [30] and separates AVIST into three aspects: data transformations (model), data views and data filters (controller); 2) data transformations, which feature the GPU parallel computation to achieve better performance; 3) coordinated multiple views, which provide different visual aspects of multidimensional datasets; 4) user interactions, which emphasize animation and cross-filtering for slicing big data into small data.

3.4.1 MVC Architecture

The implementation of AVIST follows the Model-View-Controller (MVC) architecture as shown in Figure 3.3. Filters belong to the controller, and connect with the model and view. Users can interact with one data view, then the controller updates filters, which then trigger the data models. After all data models have been updated, the controller notifies all data views to refresh visual primitives.

The MVC architecture has several benefits. 1) We separate filters, data and views, which

[3]https://developer.nvidia.com/Thrust

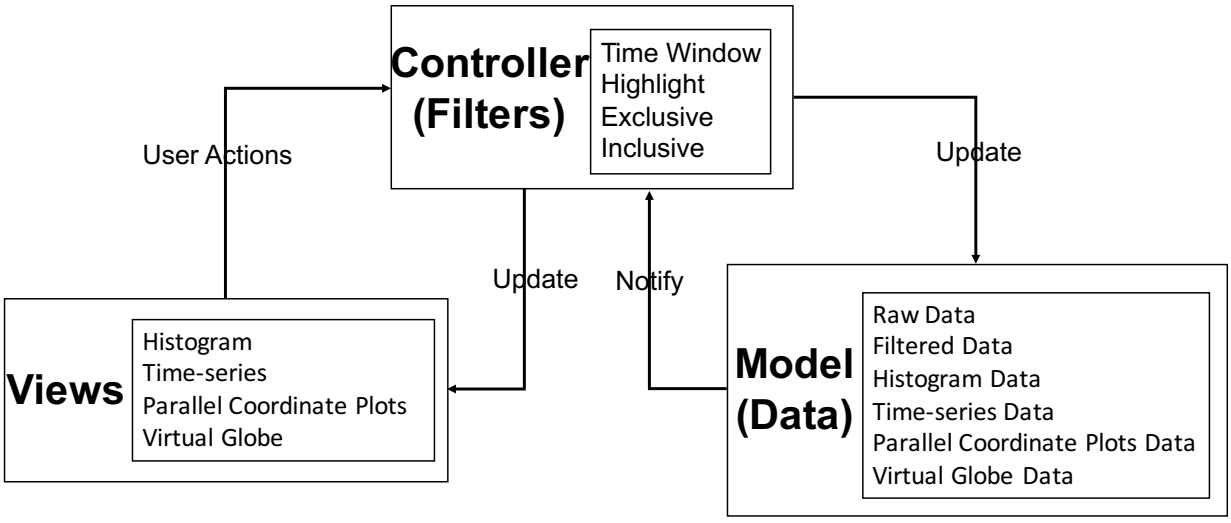

Figure 3.3: This figure shows the MVC architecture in AVIST. Users interact with data views to change data filters, so as to update data models. The controller updates the data views after it obtains the data model's notification.

gives AVIST flexibility to extend more filters, data transformations and data views in future. 2) The model is implemented by the GPU, which highlights the GPU-centric design. 3) The coordinated multiple views can be easily applied. User interactions of one data view can update filters, which trigger the data transformations and refresh other data views.

3.4.2 Data Transformation

AVIST features GPU vectorized data transformations. In this section, we describe it based on the data dependency graph design: data filtering, processing and rendering.

Data filtering

This stage describes data transformation from raw data into filtered data as shown in Figure 3.4. First, a time window slices raw data into data snapshots, then data filters are applied to data snapshots to remove uninteresting records or highlight important ones. To increase performance, a binary search is applied for slicing data based on its spatial locality. The sliced data records are checked by data filters in parallel, and then they are reduced to

a filtered dataset.

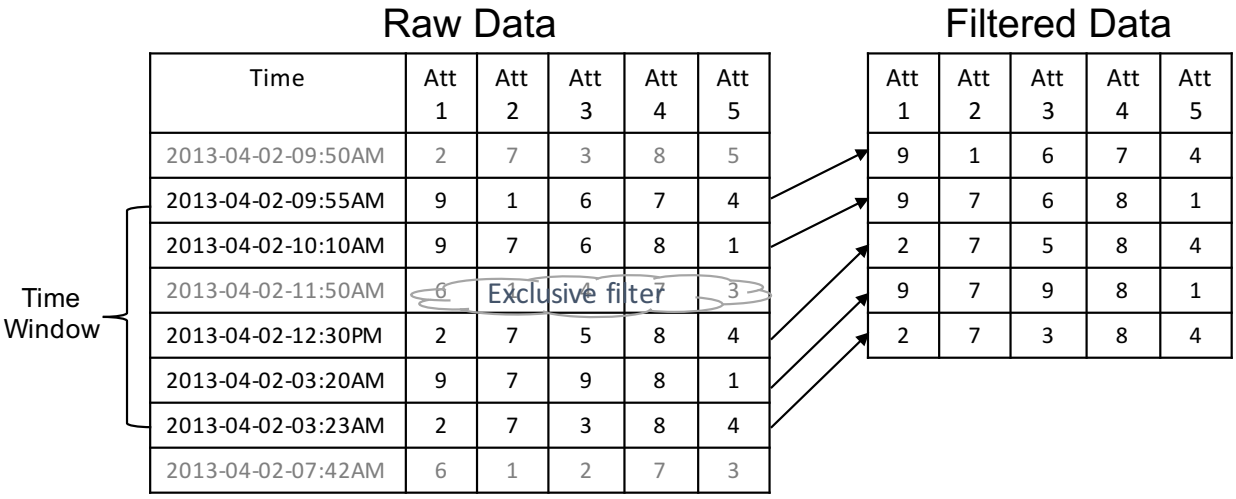

Figure 3.4: Data transformations from raw data to a filtered dataset.

Data processing

Data processing is view dependent. We demonstrate the implementation details of four data views: histogram view, time-series view, parallel coordinate plots and virtual global view (the descriptions of these are provided in Section 4.3).

Figure 3.5 demonstrates data transformation of histogram and time-series views. Based on a user's chosen column in the histogram view, AVIST retrieves the corresponding data, aggregates them, and visualizes them. The steps are described as follows.

1. Sort one data column to obtain unique values and their frequencies.
2. Get X position of each unique value based on the data range.
3. Get Y position of each unique value based on previous maximum frequency.

Data transformation of the time-series view is straightforward: the GPU counts the number of data records in a filtered dataset, then transforms it to a height value (based on the previous maximum value).

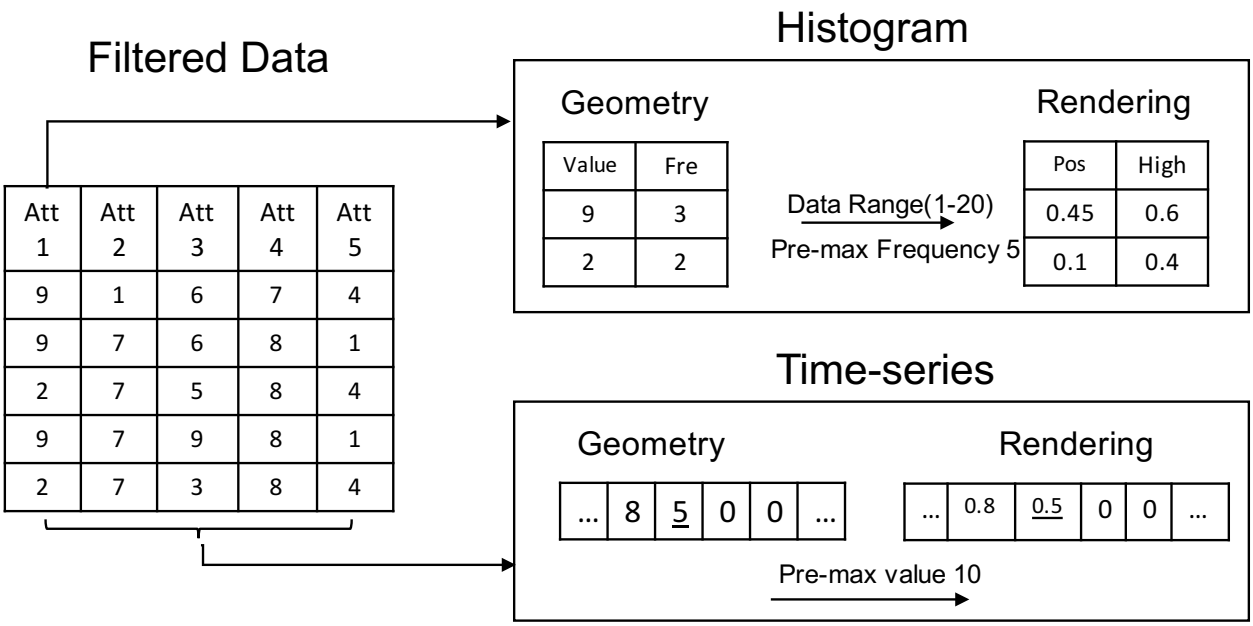

Figure 3.5: Data transformations from filtered data into visual primitives.

Data transformations of parallel coordinate plots (PCPs) are more complex than other views. First, the GPU generates PCPs compact items based on chosen axes, then data flow is separated into two parts, which includes generating vertices and edges shown in Figure 3.6. The steps for vertex generation are described as follows:

1. Sort each column individually to remove duplicated values.
2. Assemble the compact columns into three arrays: *vertex* array, *size* array and *offset* array.
3. Generate each node position: X-axis is based on each column's chosen order, and Y-axis is based on the item value and its dimension range.

The steps for generating and edge list of PCPs are as follows:

1. Two neighboring columns are grouped into one array (*Edge-ID* array) as edges. The value in this new array is calculated by:
   $Edge\text{-}ID = value_{columnA} * range_{columnA} + value_{columnB}$
2. Sort *Edge-ID* array and remove duplicates.

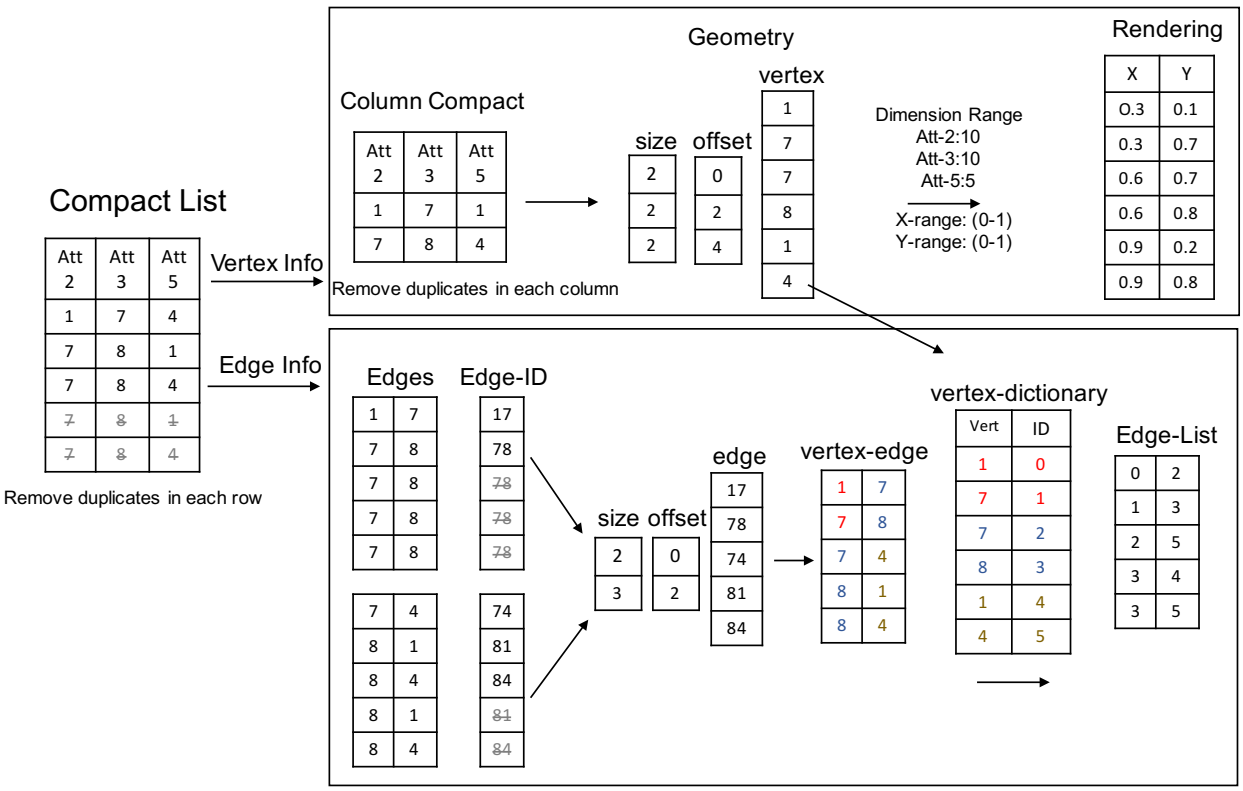

Figure 3.6: This figure shows data transformations of parallel coordinate plots, which are separated into three parts: compact list, vertex and edge generations.

3. Compact multiple *Edge-ID* arrays into three data arrays: *edge* array, *size* array and *offset* array.
4. Convert *edge* array into *vertex-edge* array.
5. Replace each vertex value in *vertex-edge* array with its order based on *vertex-dictionary* to generate *Edge-List* array.

The virtual global view is a special case of parallel coordinate plots with two axes. Their difference is that the vertex position is 3D in the virtual global view, and there are extra steps for generating 3D positions based on longitudes and latitudes. We use the Bezier curves linking two geolocations for representing their relationships.

Data rendering

After visual primitives are generated, AVIST maps them into the GPU vertex buffer objects (VBOs), which offer substantial performance gains and avoid data transfers between CPU and GPU. However, visual primitives in the GPU VBOs are vertexes and edges. In order to generate rectangles, areas or other advanced shapes, we use GLSL for post-rendering. For example, we split one vertex into four vertexes to render rectangles in histogram view. We transform vertexes into line strips to generate areas in time-series view.

3.4.3 Coordinated Multiple Views

Four data views are implemented in AVIST, which provide different visual aspects of a multidimensional dataset.

- **Histogram view** shows data distribution of a sliced data snapshot. Analysts can choose different dimensions from a listbox to explore different aggregation information.
- **Time-series view** shows data aggregation of certain filtered events over a period of time. When an analyst changes data filters, the time-series view clears previous visualization and re-draws everything.
- **Parallel coordinate plots** show details of each data record. Analysts can select multiple data dimensions to generate their custom parallel coordinate plots. The axes are organized based on their selected order.
- **Virtual global view** shows data geography distribution. A Bezier curve links two locations on the virtual global for representing their relationship.

Data views work together with data filters, which enables cross-filtering of multidimensional datasets. This design makes data views coordinated. Data views support direct visual filtering with brushing interactions (e.g., an analyst first select highlights filter with solo mode, then he brushes one bar in histogram view to highlight related data records).

3.4.4 User Interactions

AVIST is an exploration oriented visual analysis tool, which highlights animation and cross-filtering for exploring time-series and multi-dimensional datasets. Figure 3.7 shows the control panel of AVIST, which is about animation and filtering interactions.

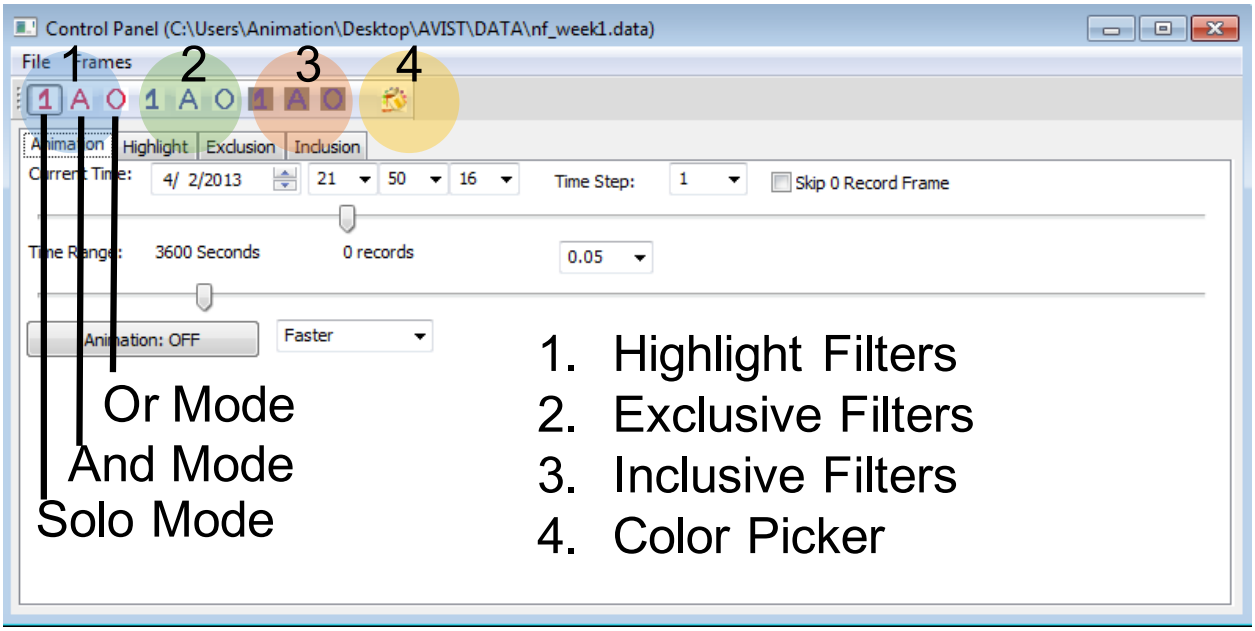

Figure 3.7: The control panel of AVIST. Three filters (*highlight filter, exclusive filter* and *inclusive filter*) with three modes (*solo* mode, *and* mode, and *or* mode) provided.

Animation

The animation interactions include automatic forward playback, interactively dragging of a time window bar, and interactive change of animation speed. By changing the current time and time range, the time window is customized to slice data into snapshots, which will be analyzed and visualized in correlated data views. By combining automated animation with these correlated views, AVIST can provide temporal changes in the datasets, which support discovery of temporal patterns for further analysis.

Filtering

Three different filters are implemented in AVIST (their usabilities are shown in following case studies).

1. *highlight filters*, which make selected data items stand out of the rest with different colors.
2. *exclusive filters*, which remove uninteresting data items.
3. *inclusive filters*, the exact opposite of exclusive filters, which remove all data items except those marked by an analyst.

Each data filter has three modes:

1. *solo* mode, which allows only one filtered item in current filter set.
2. *and* mode, which combines several filters together as a whole, to emphasize that data records satisfy all conditions.
3. *or* mode, which selects data records that meet at least one of all filter conditions.

Based on these three basic filters with their three modes, analysts can generate complex nested data filters and apply them to different data views, which will help them drill down and "find a needle" on demand.

Capacities

The animation interactions can slice high velocity data into fine-grained subsets by narrowing down the size of the time windows; filtering interactions can extract relevant records for analysts concentrating small subsets of high volume data. The cross-filtering can be achieved by combing different filters and multiple data views. These interactions transform large data into small data by removing uninteresting items or highlight important ones, which ensures the flexibility of exploring large multidimensional datasets. In all, AVIST highlights animation and cross-filtering for exploring large volume and high velocity datasets.

3.5 Performance

We test the performance of AVIST on a desktop computer, running Windows 7 Enterprise, which is equipped with an Intel i7 processor and a NVIDIA GeForce GTX 680 graphics card with 4GB memory. We use a network traffic dataset from the first case study as our benchmark.

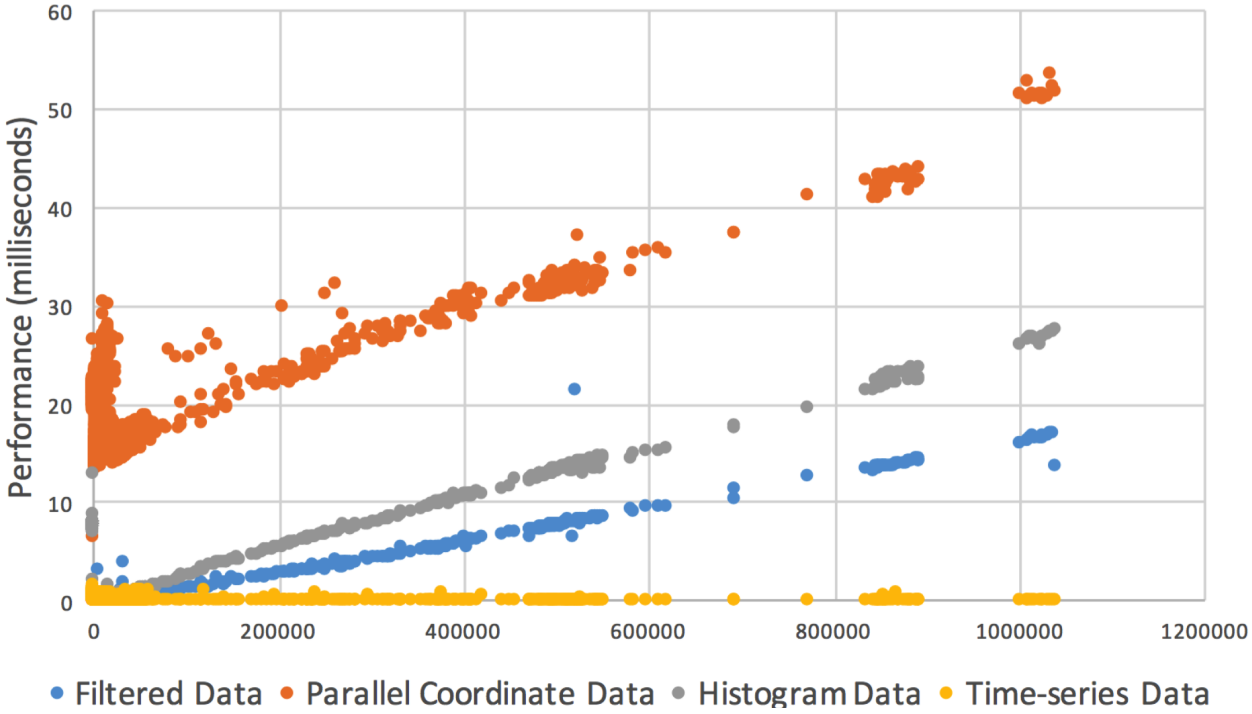

Figure 3.8: The performance of AVIST. The X-axis is the number of queried data records, and the Y-axis is the performance (milliseconds). This scatter plot shows the performance of our kinds of data: *Filtered Data*, *Parallel Coordinate Data* (eight axes), *Histogram Data* and *Time-series Data*.

We characterize AVIST performance in two stages. The first stage is about filtered data generation, which is shared by all data views. The second stage is about each data view's visual primitive generation, and their performances which are independent from each other. Figure 3.8 shows the tested performance in a scatter plot, which includes four types of data. They all show a linear relationship between the performance and number of queried data records with the exception of time-series data. The reason is that the aggregated information in time-series view does not require lots of computations. The data processing of generating

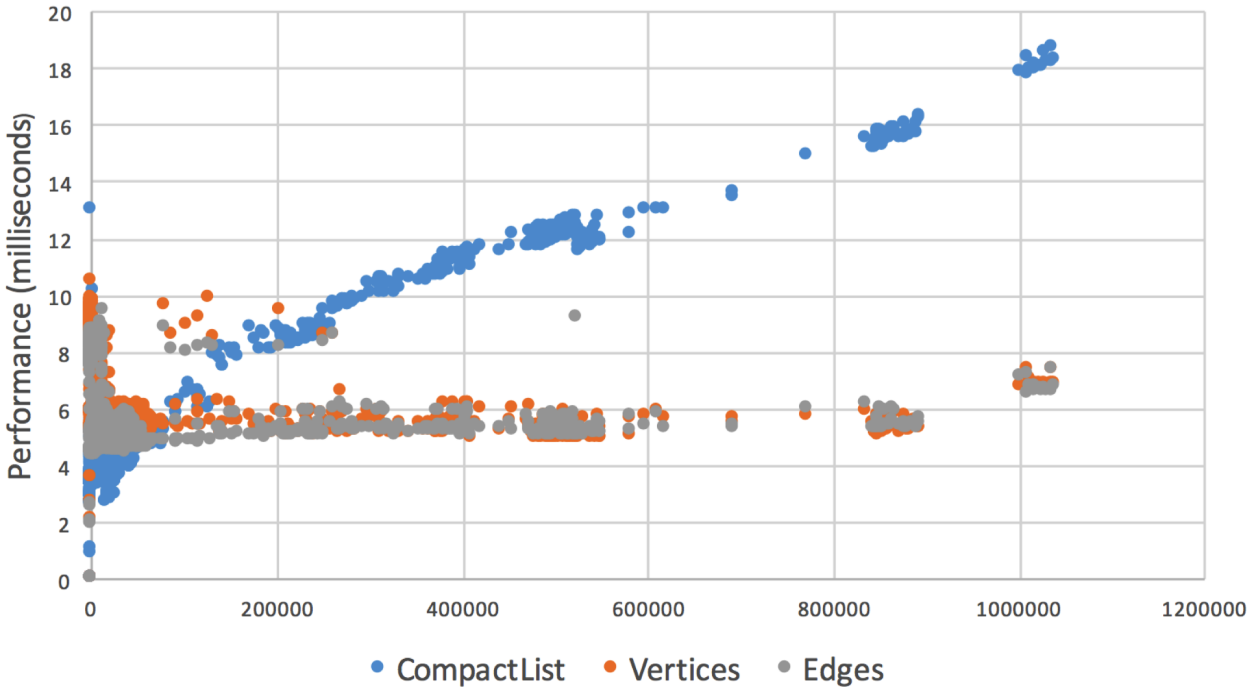

Figure 3.9: The detailed performance for generating parallel coordinate data. The X-axis is the number of queried data records, and the Y-axis is the performance (milliseconds). Three steps are characterized in the plot, which includes the data generation of compact list, vertices and edges.

parallel coordinate data is the most time consuming part in AVIST. To further investigate the detailed GPU performance in parallel coordinate plots (eight axes), we characterize data transformations into three stages, which include: 1) compact list generation; 2) vertex generation and 3) edge generation. The performance of these stages is shown in Figure 3.9, where the data generation of the compact list spends more time that the other two stages combined. The reason is that after removing duplicated rows while generating compact list, the data records used for generating vertices and edges are heavily reduced. The performance of parallel coordinate data generation is also impacted by the number of axes. Figure 3.10 shows the performance of parallel coordinate data with 2, 4, and 8 axes. We see that the performance decreases with the increased axes, but the number of columns in parallel coordinate plots has limited impact.

We have implemented the CPU version for parallel coordinate plots. The comparison between CPU and GPU is shown in Figure 3.11. We see that the GPU version is much faster

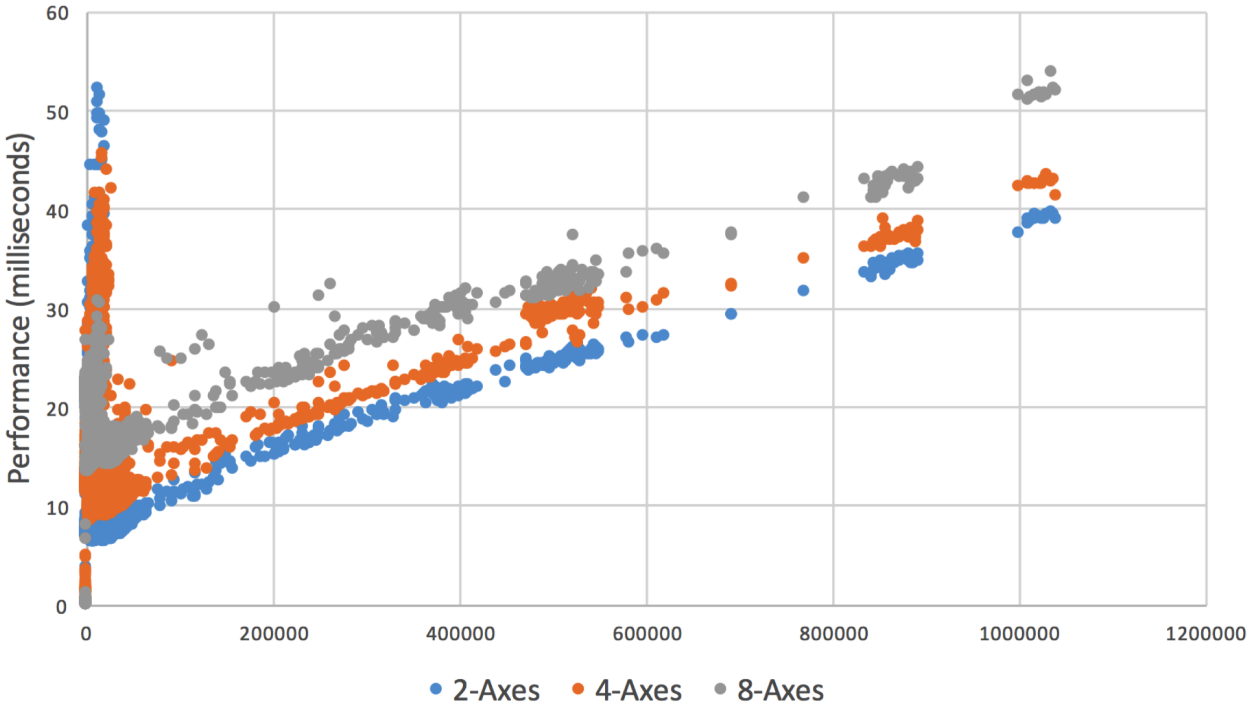

Figure 3.10: The performance for generating parallel coordinate data with different number of axes. The X-axis is the number of queried data records, and the Y-axis is the performance (milliseconds). The figure includes three cases: 2 axes, 4 axes and 8 axes.

than the CPU version. The reason is that our implementation is heavily based on removing duplicates. We have different solutions to achieve this on the CPU and GPU platforms. In the CPU version, we use the *map* and *set* containers from the C++ Standard Template Library (STL) to remove the duplicated data items. Their implementation is based on a red-black tree [51]. Thus, the time complexity of inserting N items into a *set* container to remove duplicates is $O(Nlog(N))$. In the GPU version, we use the sort operation from the Thrust library to remove duplicates. Its implementation is based on the radix sort algorithm [48]. The complexity of it is $O(NK)$, where K is the number of bits in one element (the value of K is 64 in our implementation).

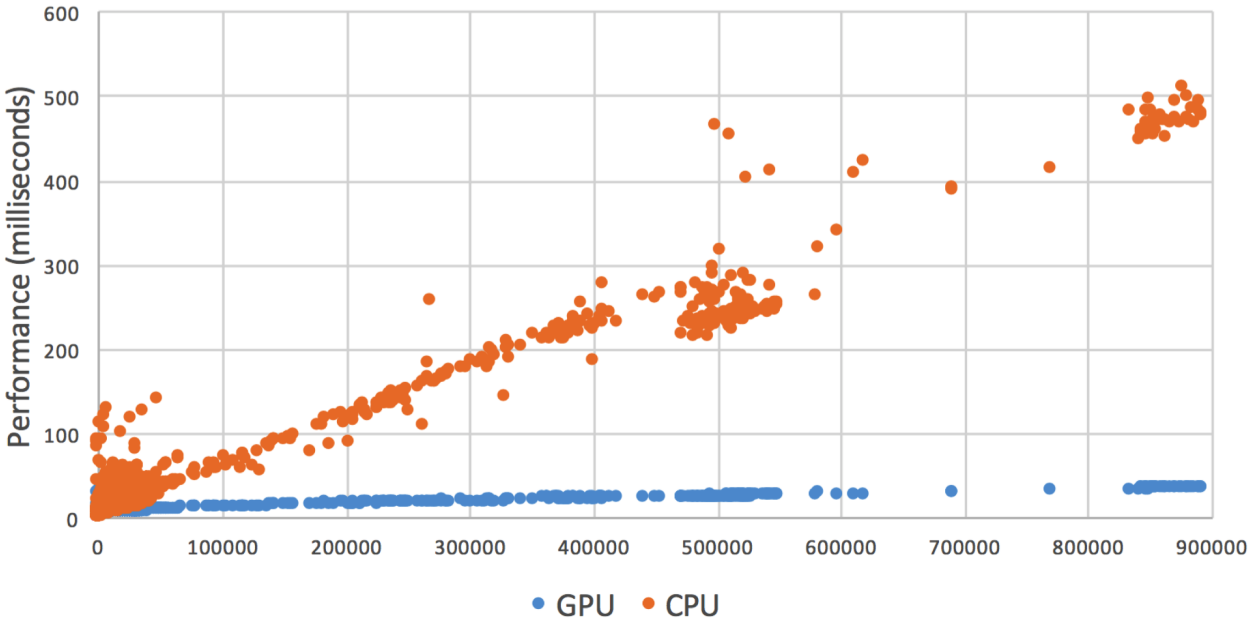

Figure 3.11: The performance comparison between CPU and GPU in parallel coordinate plots (two axes). The X-axis is the number of queried data records, and the Y-axis is the performance (milliseconds).

3.6 Usage Scenario

In this section, we use two usage scenarios to demonstrate how AVIST can support visually exploration of big data, to identify hidden patterns and infer and verify new hypothesis by analysts.

3.6.1 Network Flow Analysis

Exploratory visual analysis of network logs is critical for detecting potential cyber threats and network intrusions. VAST 2013 Mini Challenge 3, which targets the analysis of Big Marketing company networks, provides big network traffic logs[4]. One of the datasets is about one weekly network flow, which includes 46,138,310 records and 16 dimensions.

Suppose that Alice is an intelligence analyst. She is assigned to identify network threats for a Big Marketing company. At first, she preprocesses this dataset to obtain the binary data

[4]http://vacommunity.org/VAST+Challenge+2013

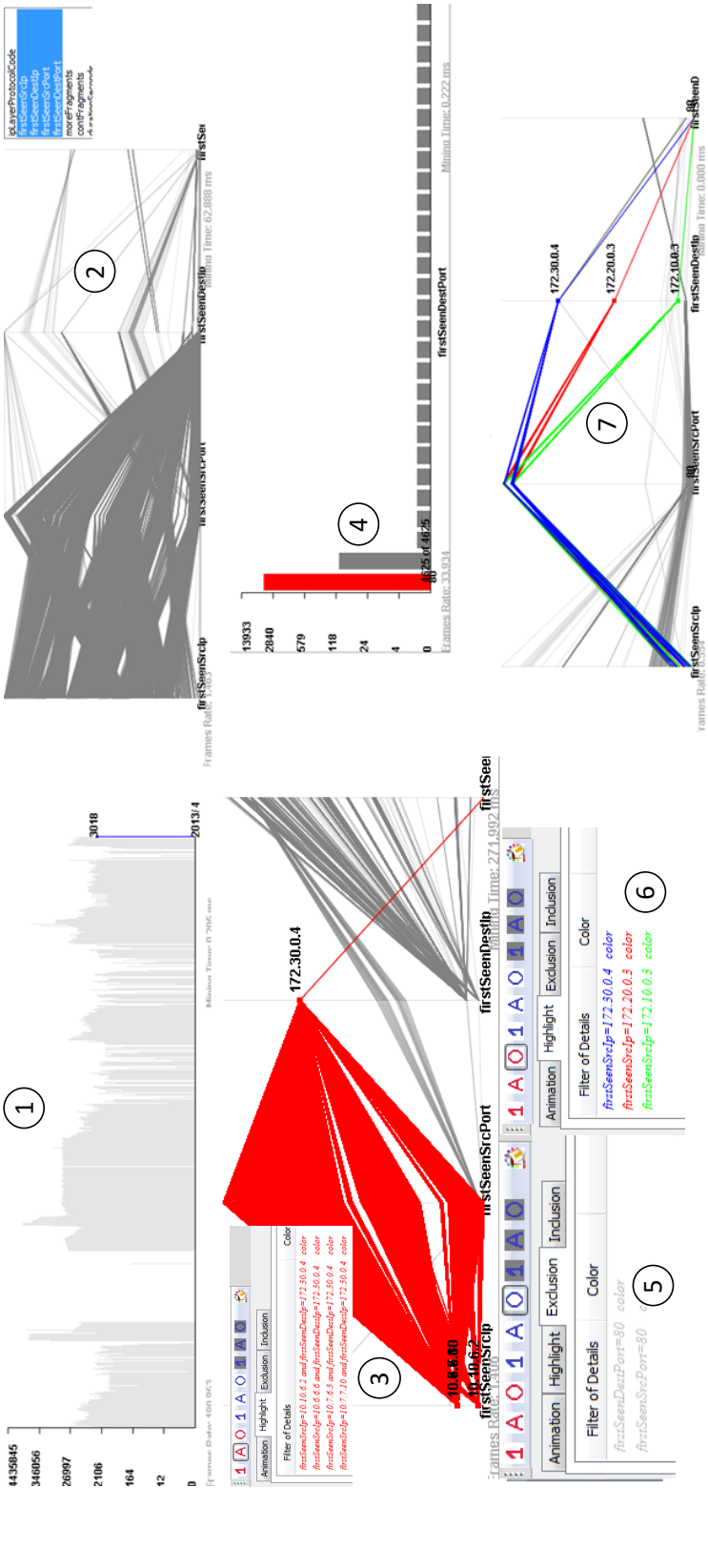

Figure 3.12: This figure shows a usage scenario how AVIST helps an analyst for visual exploration of the large network logs. Seven steps are presented in this figure, and Section 6.1 gives more detailed explanations.

and opens AVIST to load it. Figure 3.12 shows key steps of Alice's visual analytical process.

Data exploration from overview to details. At the beginning, Alice specifies the time window size (120 seconds in this example) for automatic animation. The overview of network traffic is visualized in the time-series view, shown in subgraph 1. Alice identifies an unusual behavior. The network crashed from 4-2-2013 9:40 am to 4-3-2013 3:26 am. She wants to investigate the details preceding the network crash, so she chooses data dimensions in the order of *firstSeenSrcIp*, *firstSeenSrcPort*, *firstSeenDestIp* and *firstSeenDestPort* to generate parallel coordinate plots in subgraph 2. Then she plays the animation again and discovers that four source IPs 10.10.6.2, 10.6.6.6, 10.7.6.5, 10.7.7.10 scan the destination IP 172.30.0.4 during 4-2-2013 5:20 am in subgraph 3.

Flexible filtering for revealing hidden patterns. Subgraph 4 shows the network traffic distribution of *firstSeenDestPort*. Alice realizes that most of the network records are related with port 80, and she hypothesizes that these are normal transactions. She removes these data records with port 80 (subgraph 5) to investigate buried patterns. Then she plays the animation again and discovers port scan activity shown in subgraph 7. She captures this behavior by highlight filters as shown in subgraph 6.

In this example, Alice iteratively plays animations and tries different filters for identifying potential network threats. She can easily constrain the time and data attributes to capture any subtle and hidden relationships. For example, Alice identifies the port scan activity by removing network flow related traffic with destination port 80 in subgrapah 7. However she cannot identify this activity from subgraph 2 in the first round animation. In contrast, imMens preaggregates data before visualization, and as a result it cannot provide such flexible interactions, such as cross-domain filtering.

3.6.2 International Trade Analysis

Making sense of international trade is important for governments and policy makers. With the world economic growth, world trade transactions have became large and complex. In this case study, we obtain big international trade transactions from PIERS Global Intelligence

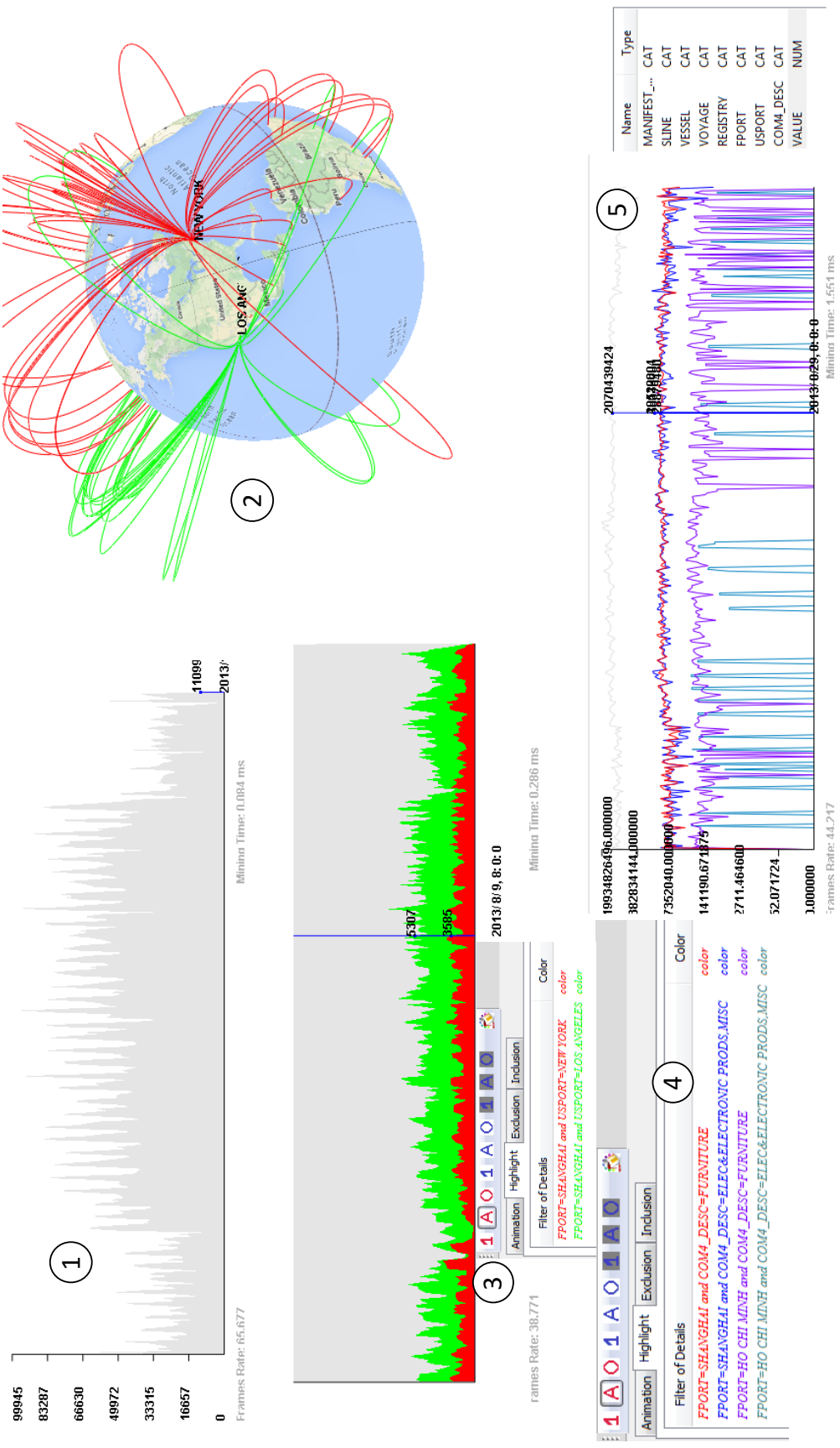

Figure 3.13: This figure shows a usage scenario that an analyst uses AVIST for visual exploring big international trade transactions. The detailed discussions are presented in Section 6.2.

Solutions[5], a private company that provides portfolio analysis for U.S. waterborne trade activity. The dataset includes about 2013 US waterborne imports 10,735,092 records and 10 dimensions of information. Within 10 dimensions, each data record features a US port code and a foreign port code, which illustrates its geospatial relationship.

Suppose that Mike is an economic analyst. He wants to gain insights from the international trade activities. So he decides to explore this dataset using AVIST. Figure 3.13 shows the insights from Mike's visual exploration.

Overview exploration. Mike specifies the time window (86400 seconds or one day) before playing the animation. The time-series view shows the overview of the world trades in subgraph 1, which presents activities that are clearly separated into four quarters. The second and third quarters have more transactions than others.

Hypothesis generation and verification. Mike hypothesizes that the international trades may have some spatial patterns. International traders prefer neighboring counties. To verify this, Mike highlights two US ports: Los Angeles (LA) and New York (NY), then he plays animation. Subgraph 2 shows one snapshot in virtual globe view, which indicates that LA has more trades with pacific countries while NY is strong related with Western Europe. Mike narrows down the data records, and he focuses on the trades from ShangHai (SH) to LA and NY. Then he plays animation to refresh the time-series view in subgraph 3. He finds that the trades from SH to LA are roughly twice of these of SH-NY, which supports his hypothesis.

Making sense of big data. The international trades can characterize each country's economy. Mike generates four filters to compare the economies of China and Vietnam. He chooses ShangHai port and Ho Chi Minh port to represent the two countries. He emphasizes the trades related to furniture and electronics, as shown in subgraph 4. He is more interested in the money value rather than the number of trades, so he clicks the value in the listbox and plays the animation. Subgraph 5 is the time-series view with four highlighted line charts, which shows that the two kinds of trades are more balanced in China, while Vietnam relies more on furniture exports.

[5]https://www.piers.com/

3.7 Lessons Learned

In this section, we summarize lessons learned for designing and implementing AVIST. At first, we discuss the trade-offs of choosing different techniques towards designing big data visual analytical tools.

- *Pre-aggregation vs Aggregation on demand*: imMens [45] is a pre-aggregation big data visual querying system based on the data cube technique. However, it is constrained by 1) long preprocessing time, 2) huge memory requirements (derived data may be larger than original data), and 3) lacking flexible filtering and querying. In contrast, AVIST emphasizes data aggregation on demand. To achieve this, it features the GPU-centric design to increase performance. Hence, AVIST can provide flexible filtering interactions. Actually, both of them can handle big data successfully. While they emphasize different aspects, the pre-aggregation technique reduces big data into small data to achieve high level data exploration. The aggregation on demand summarizes small portion of data by filtering and animating to obtain low level data details.

- *Row-Oriented vs Column-Oriented*: These are two basic data management methods, and both of them have pros and cons. In the "big data" era, column based methods gain more attention, because columnar databases have two features. First, it has better compression ratio by storing similar things together, and it reduces IO cost during data transfers from disk to memory. Second, it supports high level analytical workloads very well. While row based databases are better for on-line transaction processing (OLTP) applications, which support frequently reading and writing small transactions. The main advantage of row-oriented data is efficiently retrieving small data to find a "needle in a haystack". In our work, we favor the row-oriented method considering our exploration orientated tasks (e.g., gaining fine-grained data details).

- *Exploration-Driven vs Analytical-Driven*: Exploration driven applications emphasize "unknow-unknow" knowledge discovery, and highlight data exploration and interaction techniques. However, analytical driven applications focus on "know-unknow" insight

synthesizing, and they care about integrating statistics and machine learning techniques with human interactions and visual representations. AVIST is an exploration driven tool targeting visually exploring time series and multi dimensional datasets, and we present its design and implementation from data management, computation and exploration aspects. Thus users can quickly explore data, gain insights and generate and verify hypotheses.

- *Vertical Scaling vs Horizontal Scaling*: Performance is a key concern for designing big data visual analytical systems. Parallelization is a choice to reduce performance overhead and scale to larger datasets. In this work, we emphasize the performance gains on the GPU, in keeping with our design goal of fully exploiting the GPU resources. However, our GPU-centric design is constrained by the limited GPU resources. To handle even bigger datasets, distributed systems can be an option. However, distributed systems are designed for long batch jobs. Nowadays, those systems have focused their attention to interactive analysis tasks, and have redesigned their frameworks, such as Spark [68]. However, most of them are analytical driven systems, which emphasize scaling data mining and machine learning from small data into big data, rather than exploring large datasets.

Considering the implementation of AVIST, we have shown the GPU based data transformations. Lessons learned from implementing the GPU based visual exploration systems are organized as follows.

- *Data Management on the GPU:* The GPU is a compute-intensive processor. It is designed to perform data-parallel computation, in which a single instruction works over a large block of data. Working in blocks of data is more efficient than working with a single cell at a time. This is because the GPU reduces the overhead for decoding the instructions. A large block of parallel working units means high throughput computation. This kind of throughput-oriented computation needs high bandwidth data access. Thus, the GPU favors array based data structures.

  When programming on the CPU, programmers can write to any location in memory at

any point in their program. However, when programming on the GPU, programmers access the GPU memory in a much more structured manner. GPU is a SIMD (single-instruction multiple-data) architecture, which ensures that the computation on one data element cannot affect another. The only values that can be used in a GPU kernel are kernel input parameters and the GPU global memory reads. In addition to the above, the output of a GPU kernel should be independent. GPU kernels cannot have random writes into the GPU global memory, each thread of a GPU kernel may perform writes to a single element in the output memory.

Considering details of the GPU memory organization, programmers need to pay special attention towards data memory layout. The GPU memory has its own address space (unified memory is provided since CUDA 6.0), which means that programmers must explicitly copy data into the GPU memory before executing the GPU kernels. Compared to the CPU, the GPU lacks of implicit cache optimization. Programmers need to explicitly cache heavily used variables in shared memory to increase performance. Shared memory is limited in term of size, and inappropriate usage of shared memory may hamper the total performance, such as bank conflicts [29]. Besides, programmers should take care of how to access GPU global memory efficiently. GPU global memory features combining multiple memory accesses into as few transactions as possible to minimize bandwidth, which is also known as coalesced memory access. Thus, programmers should be aware of improper GPU memory access patterns (e.g, memory is not sequential, memory access is sparse, and misaligned memory access, etc.).

In AVIST, we carefully consider such pitfalls for the GPU programming. Our data structures are based on data arrays and we have removed duplicated data in those data arrays to reduce computation overhead. We compact multiple small data arrays into one array to access memory in a sequential way.

Designing an efficient GPU kernel is nontrivial, especially considering the GPU's memory organization. Instead of striking for complex GPU kernels, we implement the data transformations based on existing GPU libraries (e.g, Thrust). In AVIST, we use the sort operation to eliminate duplicates and obtain unique values; we use the reduction

operation for duplicate removal and data array re-organization. Based on these, we can achieve the best result for system programming and performance.

- *Data Visualization on the GPU:* We highlight the usage of the GPU VBOs for performance gains of big data visualizations. The basic visual primitives of the GPU are vertices or triangles. To generate complex visual primitives (e.g., bars and areas) or obtain advanced visual features, we use GLSL for programming on the GPU, which transforms GPU geometric primitives into a raster image.

  The GPU features its rendering pipeline with three programmable shader stages: vertex shader, geometry shader and fragment shader. Marroquim et al. [47] provided a detailed description of each stage. In fact, the graphics pipeline is independent across stages, which makes rendering performance significantly improved. Therefore, the GPU is a powerful and necessary weapon for visualization big data.

- *Data Interaction on the GPU:* The data interaction of AVIST emphasizes highlighting, brushing, filtering, and so on. These interactions include two stages for the GPU based systems.

  1) Keyboard and mouse events are collected in the CPU side, and they are transformed from the windows 2D space into the image space based on the viewport setting of the GPU. After that, these events are assembled and transfered to the GPU side.

  2) The GPU parses these events and links the visual primitives back to original data. To achieve this, the GPU VBOs unmap visual primitives back to the GPU global memory. Then the GPU filters each visual primitive in parallel so as to obtain the filtered ones. After that, the GPU traces back to original data based on the filtered ones and continues data transformations for next frame.

In all, designing GPU based high performance visual analysis systems requires plentiful GPU knowledge, which may become a high bar for researchers in the InfoVis area. Interactively InfoVis of big data needs the collaborations from multiple disciplines (e.g., graphics, database, high performance computing). Therefore, big data visualization requires InfoVis researchers

to team up with experts from other domains, so as to narrow the gap between big data and its insights.

3.8 Summary

In this chapter, we contribute a GPU-centric design for visual exploration of big data. We emphasize the GPU platform for storing and processing big data. In addition, we propose a data dependency graph design to characterize data computations on the GPU.

As a proof-of-concept, we implement AVIST based on our design. We highlight animation and cross-filtering interactions for slicing big data into fine details, we also parallelize computations for generating visual primitives. The implementation of AVIST is based on the MVC architecture, and it features the GPU data transformations of four data views.

We demonstrate how AVIST help analysts to gain insights of big data within two usage scenarios. We summarize the lessons learned, which discuss the trade-offs of different techniques for designing big data visual analytical systems. We also summarize the pitfalls and nuggets for implementing such GPU based systems. We call for interdisciplinary collaborations for big data visual analytical solutions.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

This thesis discusses the GPU based solutions for visual exploration of big data in InfoVis area. Two common data types are considered: graphs and multidimensional datasets.

To interact with big graphs, We leveraged a topology oriented force-directed layout algorithm, which calculates approximated repulsive force based on a graph topology instead of its nodes' spatial distribution. We designed a GPU friendly data structure and fine-grained parallel algorithm for deploying the proposed algorithm on the GPU. We evaluated its performance and demonstrate the interactivity in two usage scenarios.

We proposed a GPU-centric design for visual exploration of large multidimensional datasets. The design features that data storage and computation are both on the GPU for utilizing its fine-level parallelism and high memory bandwidth. A data dependency graph is also proposed to describe the GPU computation. As a proof-of-concept, an animated visualization tool (AVIST) is implemented. Two case studies demonstrate that AVIST can help analysts flexibly filter data for exploring data details.

4.2 Discussion

In this thesis, we focused on the GPU based solutions for visual exploration of big data in InfoVis area. We presented two applications for visualization of big graphs and multidimensional datasets.

Although the GPU based visualization solutions of graphs and multidimensional datasets are very different, both of them share some common features.

1. **Data storage, computation and visualization are on the GPU.** Both graphs and multidimensional datasets are stored on the GPU memory. The data storage is on the GPU, which utilizes its high memory bandwidth and avoids data transfers between CPU and GPU. The performance challenges from big data visualization systems are mainly about the data computation and rendering. The data computation leverages the GPU parallelism for increasing performance. It includes data layout computing (graphs) and visual primitive generation (multidimensional datasets), both are the general-purpose computing on the GPU (GPGPU). The rendering is based on the GPU vertex buffer objects, which can offer substantial performance gains. In other words, GPUs become a platform that unifies data storage, computation and visualization to provide a dependent solution for big data visualization.

2. **Fine-level parallelism.** GPU is a many-core architecture, and it features compute-intensive and highly parallel computation. To leverage GPU resources, we need design our own parallel algorithm (graph layout) or utilize existing GPU libraries (the exploration of multidimensional datasets). However, we need carefully design GPU friendly data structures for both cases, to remove data dependency and support fine-level parallelism.

How to utilize the GPU for big data visualization in the InfoVis area has no clear answers. However, several factors need be considered in InfoVis big data visualization, and there are many differences when compared to the SciVis.

1. **Data domain.** The data domain of SciVis includes vectors, scalar, volume, cloud points, etc. All of them have clear 3D spatial positions. The GPU can easily store and access these data. For example, the GPU texture memory has hardware optimization for retrieving its data based on the spatial locality. However, the data domain of InfoVis contains various kinds of data, such as graphs, texts, images and tabular data. Organization of such data on the GPU is a challenge. Thus, the data processing and data structure design are the keys for the GPU based InfoVis solutions.

2. **Data visualization.** The visual primitives of SciVis are about isosurface, contour, volume, etc. GPU has been optimized for generating them. For example, GPU features 3D hardware trilinear interpolation for isosurface rendering. In contrast, the visual primitives of InfoVis include points, lines, areas and so on. It is not straightforward to design GPU based algorithms for generating such big and complex visual primitives, Therefore, the visual primitive generation should be carefully considered in InfoVis big data visualization.

3. **Data interaction.** The visualization of SciVis are usually 3D, while InfoVis prefers 2D visualizations. Therefore, the data interaction techniques are very different between them. SciVis highlights user interactions including rotation, zooming in and out. Their implementations are about matrix manipulations, which can be easily parallelized on the GPU. However, user interactions in InfoVis contain brushing, filtering and highlighting. These interactions cannot be directly parallelized on the GPU. Thus how to design GPU algorithm to support fast brushing and filtering becomes an important consideration.

4.3 Future Work

In future, I would like to explore the statistics and data mining algorithms on the GPU. Data exploration is the first step for understanding big data. The next step is the interactive data mining, which is a key for modeling and quantifying big data. I look forward to see such techniques in future.

Bibliography

[1] Tableau software http://www.tableau.com/.

[2] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980, 2008.

[3] David Auber and Yves Chiricota. Improved Efficiency of Spring Embedders: Taking Advantage of GPU Programming. In *The Seventh IASTED International Conference on Visualization, Imaging and Image Processing*, VIIP '07, pages 169–175, Anaheim, CA, USA, 2007. ACTA Press.

[4] M Balsa Rodríguez, E Gobbetti, JA Iglesias Guitián, M Makhinya, F Marton, R Pajarola, and SK Suter. State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. In *Computer Graphics Forum*, volume 33, pages 77–100. Wiley Online Library, 2014.

[5] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[6] Gereon Bartel, Carsten Gutwenger, Karsten Klein, and Petra Mutzel. An experimental evaluation of multilevel layout methods. *Graph Drawing*, pages 80–91, 2011.

[7] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. *GPU computing Gems Emerald edition*, page 75, 2011.

[8] John Canny and Huasha Zhao. Big data analytics with small footprint: Squaring the cloud. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–103. ACM, 2013.

[9] Yong Cao, Reese Moore, Peng Mi, Alex Endert, Chris North, and Randy Marchany. Dynamic analysis of large datasets with animated and correlated views: VAST 2012 Mini Challenge# award: Honorable mention for good use of coordinated displays. In *IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 283–284. IEEE, 2012.

[10] Matthew Chalmers. A Linear Iteration Time Layout Algorithm for Visualising High-dimensional Data. In *Proceedings of the 7th Conference on Visualization '96*, page 127, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[11] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (OGDF). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2011.

[12] Thomas H Davenport, Paul Barth, and Randy Bean. How big data is different. *MIT Sloan Management Review*, 54(1), 2013.

[13] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, 1996.

[14] Peter Eades. A heuristics for graph drawing. *The Publisher of Congressus numerantium*, 42:146–160, 1984.

[15] Alex Endert, M Shahriar Hossain, Naren Ramakrishnan, Chris North, Patrick Fiaux, and Christopher Andrews. The human is the loop: new directions for visual analytics. *Journal of intelligent information systems*, 43(3):411–435, 2014.

[16] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[17] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V Sander, and Ke Yang. Parallel data mining on graphics processors. *Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS08-07*, 2, 2008.

[18] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *IEEE Symposium on Information Visualization*, pages 117–124. IEEE, 2002.

[19] Patrick Fiaux, Maoyuan Sun, Lauren Bradel, Chris North, Naren Ramakrishnan, and Alex Endert. Bixplorer: Visual analytics with biclusters. *Computer*, (8):90–94, 2013.

[20] Yaniv Frishman and Ayellet Tal. Multi-Level Graph Layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, November 2007.

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

[22] Pawel Gajer and Stephen G. Kobourov. GRIP: Graph Drawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.

[23] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C Hart. Rapid multipole graph drawing on the GPU. In *Graph Drawing*, pages 90–101. Springer, 2009.

[24] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[25] Leslie Greengard and Vladimir Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 73(2):325–348, December 1987.

[26] Stefan Hachul and Michael Jünger. Drawing Large Graphs with a Potential-field-based Multilevel Algorithm. In *Proceedings of the 12th International Conference on Graph Drawing*, GD'04, pages 285–295, Berlin, Heidelberg, 2004. Springer-Verlag.

[27] Stefan Hachul and Michael Jünger. An Experimental Comparison of Fast Algorithms for Drawing General Large Graphs. In *Proceedings of the 13th International Conference on Graph Drawing*, GD'05, pages 235–250, Berlin, Heidelberg, 2006. Springer-Verlag.

[28] David Harel and Yehuda Koren. Graph Drawing by High-Dimensional Embedding. *The Journal of Graph Algorithms and Applications*, 8(2):195–214, 2004.

[29] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[30] Iwao Hatanaka and Stephen C Hughes. Providing multiple views in a model-view-controller architecture, July 20 1999. US Patent 5,926,177.

[31] Jeffrey Heer and Danah Boyd. Vizster: Visualizing online social networks. In *IEEE Symposium on Information Visualization*, pages 32–39. IEEE, 2005.

[32] Jeffrey Heer and George G Robertson. Animated transitions in statistical data graphics. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1240–1247, 2007.

[33] Nathalie Henry, Jean-Daniel Fekete, and Michael J McGuffin. NodeTrix: a hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1302–1309, 2007.

[34] Yifan Hu. Algorithms for visualizing large networks. *Combinatorial Scientific Computing*, pages 1–25, 2011.

[35] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 277–281. ACM, 2015.

[36] Stephen Ingram, Tamara Munzner, and Marc Olano. Glimmer: Multilevel MDS on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):249–261, 2009.

[37] Toma Jeowicz, Milos Kudelka, Jan Plato, and Vaclav Snael. Visualization of large graphs using gpu computing. In *5th International Conference on Intelligent Networking and Collaborative Systems (INCoS)*, pages 662–667. IEEE, 2013.

[38] Tomihisa Kamada and Satoru Kawai. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31(1):7–15, April 1989.

[39] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Springer-Verlag, London, UK, UK, 2001.

[40] Marc Khoury, Yifan Hu, Shankar Krishnan, and Carlos Scheidegger. Drawing Large Graphs by Low-Rank Stress Majorization. *Computer Graphics Forum*, 31(3pt1):975–984, June 2012.

[41] Stephen G Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

[42] Yehuda Koren, Liran Carmel, and David Harel. ACE: A Fast Multiscale Eigenvector Computation for Drawing Huge Graphs, 2002.

[43] Alexandros Labrinidis and HV Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[44] Zhicheng Liu and Jeffrey Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[45] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[46] Noel Lopes and Bernardete Ribeiro. GPUMLib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3:355–362, 2011.

[47] Ricardo Marroquim and André Maximo. Introduction to GPU Programming with GLSL. In *Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on*, pages 3–16. IEEE, 2009.

[48] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[49] Chris Muelder and Kwan-Liu Ma. A Treemap Based Method for Rapid Layout of Large Graphs. *IEEE Pacific Visualization Symposium*, pages 231–238, March 2008.

[50] Chris Muelder and Kwan-Liu Ma. Rapid graph layout using space filling curves. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1301–1308, 2008.

[51] David R Musser, Gillmer J Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library.* Addison-Wesley Professional, 2009.

[52] Piotr Pawliczek, Witold Dzwinel, and David A Yuen. Visual exploration of data by using multidimensional scaling on multicore CPU, GPU, and MPI cluster. *Concurrency and Computation: Practice and Experience*, 26(3):662–682, 2014.

[53] Aaron Quigley and Peter Eades. FADE: Graph Drawing, Clustering, and Visual Abstraction. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 197–210, London, UK, UK, 2001. Springer-Verlag.

[54] Min Shih, Yubo Zhang, Kwan-Liu Ma, Jayanarayanan Sitaraman, and Dimitri Mavriplis. Out-of-core visualization of time-varying hybrid-grid volume data. In *IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 93–100. IEEE, 2014.

[55] Ben Shneiderman and Aleks Aris. Network visualization by semantic substrates. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):733–740, 2006.

[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[57] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[58] Maoyuan Sun, Lauren Bradel, Chris L North, and Naren Ramakrishnan. The role of interactive biclusters in sensemaking. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 1559–1562. ACM, 2014.

[59] Maoyuan Sun, Peng Mi, Chris North, and Naren Ramakrishnan. BiSet: Semantic Edge Bundling with Biclusters for Sensemaking. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):310–319, 2016.

[60] Anna Tikhonova and Kwan-Liu Ma. A Scalable Parallel Force-Directed Graph Layout Algorithm. In Jean M. Favre and Kwan-Liu Ma, editors, *2008 Eurographics Symposium on Parallel Graphics and Visualization*, pages 25–32. Eurographics Association, 2008.

[61] Ioannis G. Tollis, Giuseppe Di Battista, Peter Eades, and Roberto Tamassia. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, July 1998.

[62] Chris Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 171–182, London, UK, 2001. Springer-Verlag.

[63] Chris Weaver. Multidimensional visual analysis using cross-filtered views. In *Visual Analytics Science and Technology, 2008. VAST'08. IEEE Symposium on*, pages 163–170. IEEE, 2008.

[64] Chris Weaver. Cross-filtered views for multidimensional visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):192–204, 2010.

[65] Richard Wesley, Matthew Eldridge, and Pawel T. Terlecki. An Analytic Data Engine for Visualization in Tableau. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1185–1194, New York, NY, USA, 2011. ACM.

[66] Pak Chung Wong, Harlan Foote, Patrick Mackey, George Chin, Zhenyu Huang, and Jim Thomas. A space-filling visualization technique for multivariate small-world graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(5):797–809, 2012.

[67] Enas Yunis, Rio Yokota, and Aron J. Ahmadia. Scalable Force Directed Graph Layout Algorithms Using Fast Multipole Methods. pages 180–187. IEEE Computer Society, 2012.

[68] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[69] Hao Zhang, Maoyuan Sun, Danfeng Daphne Yao, and Chris North. Visualizing Traffic Causality for Analyzing Network Anomalies. In *Proceedings of the 2015 ACM International Workshop on International Workshop on Security and Privacy Analytics*, pages 37–42. ACM, 2015.

[70] Paul Zikopoulos and Chris Eaton. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.