Multi-Level Graph Layout on the GPU

Yaniv Frishman, *Student Member, IEEE* and Ayellet Tal

Abstract— This paper presents a new algorithm for force directed graph layout on the GPU. The algorithm, whose goal is to compute layouts accurately and quickly, has two contributions. The first contribution is proposing a general multi-level scheme, which is based on spectral partitioning. The second contribution is computing the layout on the GPU. Since the GPU requires a data parallel programming model, the challenge is devising a mapping of a naturally unstructured graph into a well-partitioned structured one. This is done by computing a balanced partitioning of a general graph. This algorithm provides a general multi-level scheme, which has the potential to be used not only for computation on the GPU, but also on emerging multi-core architectures. The algorithm manages to compute high quality layouts of large graphs in a fraction of the time required by existing algorithms of similar quality. An application for visualization of the topologies of ISP (Internet Service Provider) networks is presented.

Index Terms—Graph layout, GPU, graph partitioning.

1 Introduction

Graph drawing addresses the problem of constructing geometric representations of graphs [24,38]. It has applications in a variety of areas, including software engineering, software visualization, databases, information systems, decision support systems, biology, and chemistry.

Producing pleasing graph layouts fast is still a challenging problem. For instance, one of the most popular graph layout algorithms, the *force directed* algorithm, is computationally expensive. The complexity of each iteration of the algorithm is $O(V^2+E)$. On large graphs, the layout procedure can take anywhere from a few seconds to several minutes to complete, hindering the capability to use this algorithm to explore large data sets.

In recent years, a popular way to accelerate computations is to perform them on the *GPU (graphics processing unit)* [5,14,31,32]. This is due to the high computational power, low cost, and ubiquity of GPUs in every modern PC. GPUs are geared towards repetitively performing the same computation on large streams of data. Therefore, the GPU suits uniformly structured data, such as images or matrices. Graphs do not posses a uniform structure, hence, they do not admit any intuitive and natural representation that suits computation on the GPU.

This paper proposes two ways in which force directed algorithms can be accelerated. The first is a general multi-level scheme, which is based on spectral partitioning. The second is computation of a graph layout on the GPU.

Multi-level graph layout algorithms have been proposed in the past [9, 15, 18, 20, 26, 34, 40]. In these algorithms, the given graph is recursively coarsened, to compute its multi-level representation. In contrast, in our scheme, the algorithm works on a high-detailed graph at all levels of the partitioning. Thus, a good hierarchical representation of the graph is obtained. The scheme proposed in this paper is a general multi-level scheme, which is based on spectral partitioning. Using a bottom up approach, layouts of increasing detail are computed. It is shown how coarse layouts of a graph can be efficiently extended to the final high quality layout.

In addition, this paper describes a method of representing graphs so as to make efficient use of GPU resources. Partitioning is used to break the large problem into smaller and similarly-sized problems that suit computation on the GPU or on other data-parallel programming models. This algorithm exposes the underlying structure of the graph, and thus can be used in a multi-level scheme.

• *The authors are with the Technion, Israel Institute of Technology. E-mail: frishman@tx.technion.ac.il , ayellet@ee.technion.ac.il.*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 27 October 2007.

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

Another algorithmic contribution of the paper is devising a layout algorithm that combines the strengths of two different well-known layout algorithms [8, 23]. The produced layouts are as good as existing state of the art layouts [15, 16], yet computed at a fraction of the running time. For example, a layout of the graph bcsstk31 is computed using our approach in 5.8 seconds compared to 83 seconds in [15].

Implementation-wise, the paper elaborates on how force directed layout is accelerated, by performing the time-consuming stages on the GPU. The data storage and the stream processing are described.

Last but not least, the algorithm is applied to the visualization of the topologies of *Internet Service Providers (ISP) networks*. In this application, illustrated in Figure 1, nodes represent routers and edges represent the connections between them.

2 Related Work

Many algorithms have been proposed to perform graph layouts [24, 38]. This paper focuses on force directed layout [8,23], which is based on simulating the graph as a network of charged particles that repel each other, where edges are simulated by springs. The algorithm is popular due to its ability to draw general undirected graphs, its ability to be tailored according to specific requirements, and the aesthetically pleasing layouts it produces. However, a major drawback of the algorithm is its high computational cost.

Some algorithms have been proposed to perform force directed layouts of large graphs [16]. In [40] coarser representations of the graph are recursively built using the *edge collapse* operation. Instead of computing all-pairs repulsion forces, only close-by nodes are addressed. The algorithm in [18] creates coarse graphs using an approximation of the k-center problem. A modified version of [23] is used to perform single level layout. This algorithm requires $O(V^2)$ memory and $O(VE)$ time. The algorithm in [2] computes repulsion forces in $O(N\log N)$. In [34] a quadtree is used to accelerate layout and to visualize the graph in multiple levels of detail. In [9] a maximum independent set filtration is used to coarsen the graph. At each level new nodes are placed in accordance with their neighbors. A local force computation is performed using both [23] and [8]. FM^3 [15] is a state of the art multi-level algorithm [16]. There, solar systems are created, which consist of nodes at a distance of two edges or less from the center of the solar system. A clever $O(N\log N)$ approximation of the all-pairs repulsive forces is used to accelerate layout.

In [26] a simplified energy function is used, which allows more robust mathematical treatment. The layout problem is reduced to an Eigen value computation problem, which is solved using an algebraic multi-grid approach. Although the resulting algorithm is very rapid, the quality of the layout is limited [16]. This may be attributed to the algorithm defining forces only along edges of the graph. In [20] a high dimensional embedding of the graph is computed and then projected into the drawing plane, allowing a linear time $O(E+V)$ algorithm.

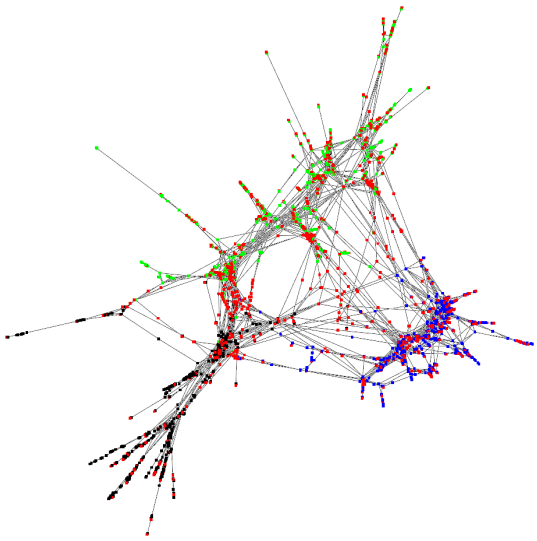

Fig. 1. ISP router map. Each node represents a router. Edges link routers. Red nodes are external to the ISPs visualized. Other nodes are colored according to the ISP they belong to: green - Abovenet (US, 664 routers); blue - Exodus (US, 551 routers); black - Tiscali (Europe, 513 routers). A total of 5044 routers and 8043 connections are shown.

In the current paper, instead of working on increasingly coarsened graphs, the input graph is partitioned to smaller and smaller parts. This helps construct an accurate multi-level representation of the graph.

In recent years, GPUs have been successfully applied to numerous problems outside of classical computer graphics [31]. Some GPU usage examples include solving differential equations [13], linear algebra [10,27], signal processing [29], visualization [17,22] and simulation [21,25,28], to name a few.

Several other GPU applications are somewhat related to ours. In [12,37] simulation of deformable bodies using mass-spring systems is performed. However, while the mass-spring algorithms take only nodes connected by edges into account, the force directed algorithm considers all the nodes when calculating the force exerted on a node. GPUs have also been used to simulate gravitational forces [30], where an approximate force field is used to calculate forces. Accelerating dynamic graph drawing on the GPU has been addressed in [7]. The focus of that work was on creating stable layouts of changing graphs, whereas the current paper addresses static layouts.

3 Spectral Graph Partitioning

Computing directly the layout of a large graph is both time-consuming and difficult. This is due to the sensitivity of force directed layout to the initial conditions given to the algorithm. To address these problems, multi-level schemes have been used [9, 15, 18, 20, 26, 34, 40]. The key idea is that a good representation of the overall structure of the graph will yield a layout of the "skeleton", which can be quickly computed, and which can assist in drawing the large input graph.

We propose an algorithm for creating a series of resolution decreasing representations of the graph by recursively partitioning it. We require the parts to have similar size and have a minimal cut between them. The former requirement helps preserve the balance between the nodes during layout, while the latter guarantees that different parts are weakly coupled and hence can be treated relatively independently.

While existing multi-level graph layout algorithms recursively coarsen the graph in order to compute the multi-level representation, our algorithm works on a high-detailed graph at all levels of the partitioning. This allows us to obtain a high-quality representation of the graph, which does not suffer from the growing inaccuracy involved in repetitively creating coarser and coarser representations of a reduced version of the graph.

To do it, we use spectral graph theory [4]. This theory has been used in the field of parallel computation to partition computation dependency graphs, where the amount of work between processors needs to be balanced [33]. It was also used in image segmentation, where *normalized cuts* were introduced [35]. The idea of using eigenvectors of the Laplacian for finding partitions of graphs has a rich history [6].

Suppose that w_{ij} is the weight of the edge (i,j), D is a diagonal matrix, $D(i,i) \equiv \sum_j w_{ij}$, and $W(i,j) \equiv w_{ij}$ is the graph edge weights matrix. The matrix $L = D - W$ is the *Laplacian* of graph G. The goal is to partition G into two equal-sized partitions A, B. For node i, we define $q_i = 1$ if $i \in A$ and $q_i = -1$ if $i \in B$. It can be shown [33] that the cut size J is:

$$J = CutSize = \frac{1}{4}\sum_{i,j} w_{ij}(q_i - q_j)^2 = \frac{1}{2}q^T(D-W)q.$$

In order to minimize J, we can relax the indicators q_i to continuous values and take the second smallest eigenvector of

$$(D-W)q = \lambda q.$$

This vector is known as the *Fiedler vector* [6]. (The smallest eigenvector, corresponding to an eigenvalue $\lambda_1 = 0$ is $q_1 = (1,...,1)^T$.)

To compute the Fiedler vector, we use the power iteration algorithm [41], shown in Figure 2. The input of the algorithm is a guess for the Fiedler vector, stored in $v2$. The computed Fiedler vector is returned in $v2$. The algorithm is iterative. In each iteration $v2$ is orthogonalized against the first eigenvector and multiplied by the matrix B which is used to reverse the order of the eigenvectors, using the Gershgorin bound, which bounds the magnitude of the largest eigenvalue of the Laplacian. This algorithm fits sparse matrices (i.e., graphs), since it requires only matrix-vector multiplications. A similar algorithm is used in [26] to directly compute the graph layout, whereas it is used here only to partition the graph.

L = Laplacian(G)
g = Gershgorin_bound(L) = $\max_i \left(L_{ii} + \sum_{j \neq i} |L_{ij}|\right)$
B = gI - L
$v1 = \frac{1}{\sqrt{N}} \cdot 1_N$ //first (known) eigenvector
do
  $v2_{old} = v2$
  $v2 = v2 - (v2^T \cdot v1)v1$
  $v2 = B \cdot v2$
  $v2 = \frac{v2}{\|v2\|}$
until $|v2_{old} \cdot v2^T - 1| < \varepsilon$ or max iteration count reached

Fig. 2. The power iteration algorithm

A drawback of the power iteration algorithm is its slow convergence rate. To accelerate the convergence, a multi-grid algorithm is used. Instead of directly operating on the largest Laplacian matrix, a series of coarsening operations is performed, until reaching a minimal problem size. The coarsening algorithm is detailed in Section 4, Step 1. After coarsening, the coarser problems are recursively solved and interpolated back, setting a good initial guess for the next (finer) problem.

After computing the Fiedler vector $v2$, it is used to partition the graph. Each node in the graph has a corresponding value in $v2$. This value is used to determine which partition the node will be assigned to. The vector $v2$ is sorted. A set of $k-1$ splitting values is determined by sampling the sorted vector at $k-1$ uniformly spaced points. This splits the vector into k regions. The partition to which a node is assigned is computed by determining to which of the k regions the value of $v2$ corresponding to the node belongs to.

Since the graph is partitioned into more than two parts, some clusters may be disconnected. A post-processing stage that merges clusters is performed. Each cluster whose size is below a threshold, is merged with its largest neighboring cluster.

The partitioning algorithm continues repetitively, building finer and finer representations of the graph. The finer representations are then used in a multi-level scheme, described in Section 4, to compute a globally pleasing layout of the original graph.

In our implementation, any eigen problem of a size smaller than 128 nodes is directly solved, since coarsening it further is not time-effective. For each problem, a maximum of 10000 power iterations are allowed and an accuracy $\varepsilon = 10^{-8}$ is used. The graph is partitioned by default into three parts ($k = 3$). Disconnected clusters smaller than $\frac{1}{9}$ of the graph are merged. Our attempts to perform a more adaptive partitioning, resulted in lower quality results.

4 Multi-level layout Algorithm

Given an undirected weighted graph $G = G^0 = (V,E)$, the goal of the algorithm is to compute a straight-line drawing of G, assigning 2D coordinates to each node. Our algorithm is based on the force-directed approach [8, 23, 24, 38], which simulates a system of forces defined on the input graph and converges towards a local minimum energy position, starting from an initial placement of the vertices.

Our algorithm has several key ideas. First, a multi-level scheme is used to compute the layout. Instead of directly computing a layout for the input graph, several coarsened versions of it are created. Starting from the coarsest version, a series of increasingly detailed layouts are computed. Care is taken to interpolate positions from each coarse layout and use them as the starting point for the next finer layout.

Second, spectral partitioning methods are used to compute lower resolution representations of the graph, as discussed in Section 3. Using this approach the difficult graph partitioning problem is transformed to a 1D partitioning problem. Breaking the graph into increasingly finer parts allows us to produce a series of increasingly detailed graphs, which are used in the multi-level scheme.

Third, a layout algorithm which combines the strengths of [8, 23] is used. While [23] is able to compute a good layout, given any starting point, it is time consuming. The algorithm of [8] is faster and computes "smoother" layouts, but is more sensitive to the initial conditions given to it. We propose an algorithm which combines the strengths of both algorithms in order to produce the final layout.

The algorithm is composed of the following stages, shown in Figure 3: We elaborate on each stage below.

1. Initial coarsening: compute $G^1, G^2, \ldots, G^{coarsest}$ where $G^{k+1} = edge\_collapse(G^k)$.
2. Partitioning initialization: set $P^{level=0}_{part\ num=0}$ to $G^{coarsest}$. Set $l = 0$.
3. Partitioning: try to partition each graph P^l_n. This creates a new set of graphs $P^{l+1}_0, P^{l+1}_1, \ldots$. If no graph P^l_n could be partitioned, goto step 7.
4. Multi-level construction: construct L^l out of $G^{coarsest}$, where each node in L^l corresponds to a graph P^l_n.
5. Layout initialization: compute an initial layout for L^l, using interpolated initial positions from the coarser L^{l-1}.
6. Layout: compute the layout for L^l. This is the core step of the algorithm, which uses our variant of the force-directed approach. Set $l = l + 1$, goto step 3.
7. Compute a layout for $G^{coarsest}$ using interpolated initial positions from L^{finest}, the finest graph layout computed in stage 6.
8. Final un-coarsening: Compute layouts for $G^{coarsest-1}, G^{coarsest-2}, \ldots, G^0$ by repetitively interpolating from G^i to G^{i-1} and laying out G^{i-1}.

Fig. 3. Algorithm overview

Initial coarsening (Step 1): In step 1, the graph is coarsened several times, as a pre-processing stage that helps reduce computation time. At each level k, given a fine graph G^k, a coarser representation G^{k+1} is constructed using a series of *edge collapse* operations [40]. A collapse operation replaces two connected nodes and the edge between them by a single node, whose weight is the sum of the weights of the nodes being replaced. The weights of the edges are updated accordingly. (The initial weight of a node/edge is 1.) The order of the edge collapse operations is different than in [40]: First, candidate nodes for elimination are sorted by their degree, so as to eliminate low-degree nodes first. An adjacent edge of a low-degree node is chosen for collapse by maximizing the following measure: $\frac{w(u,v)}{w(v)} + \frac{w(u,v)}{w(u)}$, where $w(x)$ is the weight of node x and $w(x,y)$ is the weight of edge (x,y). This function helps to preserve the topology of the graph by "uniformly" collapsing highly connected nodes.

In our implementation, three initial coarsening steps are performed. This significantly reduces the computation time of spectral partitioning (Step 3), while maintaining a good relation between the input graph G^0 and $G^{coarsest}$.

Partitioning initialization (Step 2): This step initializes the variables used in the recursive partitioning of graph $G^{coarsest}$ in the next step. The graph P^0_0, which is set to $G^{coarsest}$, is created.

Partitioning (Step 3): The goal of this step is to create high quality coarser representations of the graph $G^{coarsest}$, which are used in the multi-level layout scheme.

Starting from the single graph P^0_0 at level 0, for each level l the set of graphs P^l_n in this level are partitioned as described in Section 3. Each graph P^l_n is partitioned into graphs P^{l+1}_m, by adding the corresponding edges from P^l_n. As the level number l increases, $G^{coarsest}$ is partitioned into a growing number of graphs decreasing in size.

Multi-level construction (Step 4): A series of graphs $L^0, L^1, \ldots, L^{finest}$ of increasing detail is created. At level l, the graph L^l is created as follows. Each node n_k in L^l corresponds to a single graph P^l_k in level l. The weight of a node n_k in L^l is the sum of the weights of the nodes in graph P^l_k it corresponds to. Edges (n_k, n_j) in L^l are created by summing corresponding edges in $G^{coarsest}$ which connect the nodes in $G^{coarsest}$ corresponding to P^l_k and P^l_j.

Layout initialization (Step 5): The goal of this stage is to compute a good initial layout of L^l. This is done based on the layout of L^{l-1}, and proceeds as follows. Initially, each node $p_i \in L^l$ is placed at the position of its parent node in L^{l-1}, whose layout was already computed. Next, the position of each node is scaled, as follows:

$$p_i(x,y) = \sqrt{\frac{|V(L^l)|}{|V(L^{l-1})|}} \cdot p_i(x,y), \tag{1}$$

where $V(L^k)$ is the set of nodes in L^k. The intuition behind Eq. 1 is that the scale should be proportional to the ratio between the number of nodes in the graphs L^l and L^{l-1}. Finally, an iterative algorithm is used to improve the placement. At each iteration, each node i is placed at the average between its current position, p_i, and the average position of its neighbors, $N(i)$, as follows:

$$p_i = \frac{1}{2}\Big(p_i + \frac{1}{degree(i)} \sum_{j \in N(i)} p_j\Big).$$

This procedure creates a good initial placement, which is used in the next step. In our implementation 50 iterations are used.

Layout (Step 6): In this stage, a layout for L^l is computed, using our variant of the force directed approach. This is done utilizing the multi-level scheme, until the final layout of the finest graph, L^{finest}, is computed. Using this scheme, it is possible to retain important information about the overall structure of the graph from previous layouts, which is extracted from the spectral partitioning of the graph.

There are a couple of common approaches to performing force directed layout. The first common approach, exemplified by the Fruchterman-Reingold (FR) algorithm [8], computes the forces directly. Each node is moved according to the forces acting on it. It computes "smooth" layouts, but is sensitive to the initial conditions given to it. A second common approach, used in the Kamada-Kawai (KK) algorithm [23], derives an energy function from the forces and attempts to minimize the energy in order to create the layout. The node that reduces the energy the most is moved in each step. This algorithm is less sensitive to the initial conditions. However, it requires an expensive all-pairs shortest path calculation and the computed layouts are less "smooth".

In this paper, an approach that combines the strengths of both algorithms is used. The key idea is to use the KK approach, to give the overall structure of the graph and reduce the sensitivity to initial conditions. Then, the computed layout is used as an input to the FR-based algorithm. On finer graphs, only the faster FR layout is used. By doing so, we get a good initial placement from the KK algorithm and a "smooth", aesthetically more pleasing layout from the FR algorithm. Note that a combined approach is used in [19] in order to meet node-size constraints. In the current paper, however, FR is used to refine the layout of finer graphs in the multi-level hierarchy.

The most expensive step of the FR algorithm is the computation of all-pairs repulsive forces between nodes, which is crucial for obtaining a good layout. This step is accelerated in two ways. First, the graph is geometrically partitioned. Instead of calculating all-pairs repulsive forces, as customary, approximate forces are calculated. An exact calculation is performed only for nodes contained in the same partition, while an approximate calculation is performed for nodes belonging to different partitions. Second, the calculation of the forces is parallelized and performed on the GPU.

Graph L^l is now partitioned geometrically, according to the current layout, so as to balance the number of nodes per partition. This is important in order to achieve good load balance between the parallel processors of the GPU (Section 5). Moreover, since the nodes in each partition are geometrically localized, it is possible to approximate the partitions with a single "heavy" node, as discussed below.

Specifically, a KD-tree-type partitioning is created. The nodes are partitioned according to their median, alternating between the X and Y coordinates. This recursive subdivision terminates when the size of the subset is below the required partition size.

The algorithm is iterative. In each iteration, the KD-tree is updated according to the current layout (while required). Then, the center of gravity is found for each partition and is used to replace the nodes it contains. Next, The forces applied to each node are computed. Finally, the nodes are displaced according to the forces acting on them, while bounding the allowed displacement according to the exponential converge schedule, which resembles simulated annealing.

The key to achieving high performance is to perform these computations (i.e., finding the center of gravity of the partitions, calculating the various forces acting on the nodes, and calculating the displacements), in parallel on the GPU for each node/partition.

In particular, the repulsive and attractive forces that are computed in parallel for each node are as follows. The difference from [8] is that the forces from distant partitions are approximated using their center of gravity CG. For each node v that belongs to partition P_i,

$$F^{repl}(v) = K^2\Big(\sum_{u\neq v, u\in P_i} \frac{pos(v)-pos(u)}{\|pos(v)-pos(u)\|^2} + \sum_{P_j\neq P_i} |P_j| \frac{pos(v)-CG(P_j)}{\|pos(v)-CG(P_j)\|^2}\Big)$$

$$F^{attr}(v) = \sum_{u:(u,v)\in E} \frac{\|pos(u)-pos(v)\|(pos(u)-pos(v))}{K}$$

The attractive and repulsive forces are then summed up in parallel for every node, resulting in an approximation of the total force applied to each node, $F^{total}(v)$. Then, each node is displaced, in parallel, using a simulated annealing technique, which exponentially decreases the allowed displacement:

$$pos_{new}(v) = pos(v) + \frac{F^{total}(v)}{\|F^{total}(v)\|} min(t, \|F^{total}(v)\|).$$

Here, t is the bound for the maximum displacement, which is initialized to $K * \sqrt{|V|}$ and decreases at each iteration by a factor λ. In our implementation, $K = 0.1$ and $\lambda = 0.9$

The simulated annealing technique makes the graph slowly freeze into position. Thus, later iterations perform increasingly local corrections to the layout. Because of this behavior, it is possible to perform geometrical KD partitioning of the graph with decreasing frequency.

In our implementation, re-partitioning is done on iterations 1-4 and then every 10 iterations. A total of 50 FR iterations are performed [40]. KK layout is performed on graphs smaller than 1000 nodes. This constant was selected so the layout time will not be dominated by KK layout which requires performing an expensive all-pairs shortest path calculation. We use 2000 iterations in each KK layout.

Layout of $G^{coarsest}$ (Step 7): In this step, the layout of L^{finest} is extended to a layout for $G^{coarsest}$. Here, the same method applied in Steps 4–6, is used. Instead of interpolating positions from L^{i-1} to L^i, an initial placement for $G^{coarsest}$ is computed using the existing layout of L^{finest}. The mapping of nodes between $G^{coarsest}$ and L^{finest} is performed similarly to Step 4: each graph P_n^{finest} corresponds to several nodes in $G^{coarsest}$. After computing an initial placement for $G^{coarsest}$, layout proceeds as discussed in Step 5-6.

Final un-coarsening (Step 8): This step extends the layout of $G^{coarsest}$ to a layout of the original graph $G = G^0$. In each iteration, the layout of G^i is used to compute an initial placement for the nodes of the finer graph G^{i-1}, using the algorithm described in Step 5. Then, the force directed algorithm of Step 6 is applied to the initial placement of nodes in G^{i-1}.

In our implementation, we do not perform force directed layout of the final graph G^0, for which the layout is the most expensive. Instead, using the layout of G^1 and the interpolation algorithm for computing initial positions, we are able to get a good layout for G^0.

Complexity: The most time consuming steps of the algorithm are spectral partitioning and the FR force directed layout. Assuming that each KD partition of the graph contains C_s nodes, the asymptotic FR complexity is $O(|E| + |V| * (C_s + \frac{|V|}{C_s}))$, which is minimized to $O(|E| + |V|^{1.5})$ when $C_s = \sqrt{|V|}$. The spectral partitioning takes $O(|V|^{1.5})$ [35]. Therefore, the total complexity is $O(|E| + |V|^{1.5})$. When $|E| \approx |V|$, the dominating term is $|V|^{1.5}$. However, due to the calculation's simplicity and its parallel implementation, the actual running times are low, as discussed in Section 6.

5 GPU Implementation

This section describes how the GPU is utilized to accelerate the force-directed layout. It elaborates on key details, which are briefly introduced in [7]. Figures that illustrate the overall process are included.

The key to high performance on the GPU is using multiple processors, which operate in parallel. The GPU schedules the execution of multiple threads, thus hiding memory access latency. Each thread runs a small program called a *kernel program*, which computes a single element of the output *stream*.

In the following, we first describe how the data is stored on the GPU and then how the stream processing is performed [3].

Data Storage: On the GPU, input and output are represented as two-dimensional arrays of data, called *textures*. The challenge is to map the graph and its elements onto textures, even though graphs do not admit any intuitive and natural representation as balanced arrays. Below, we describe the textures used to represent the graph,

To represent the graph layout, three textures are used: one texture for the nodes and two textures for the edges.

The *location* texture holds the (x,y) positions of all the nodes in the graph. Each graph node has a corresponding (u,v) index in the texture. As shown in Figure 4, the nodes in each partition are stored at a rectangular region in the location texture. Recall that Section 4 described how to partition a graph, so that the nodes in each partition are geometrically close and the number of nodes in each partition is similar. This partitioning is critical for the acceleration of the layout on the GPU for two reasons. First, storing neighboring nodes (those that belong to the

same partition) together maximizes memory access locality. Thus, it makes efficient use of the GPU's memory bandwidth, since information regarding neighboring nodes will most likely reside in the cache. Second, since the number of nodes in each partition is similar, the amount of computation performed on each node is balanced. Thus, it makes efficient use of the GPU's data parallel architecture, which requires lock-step execution.

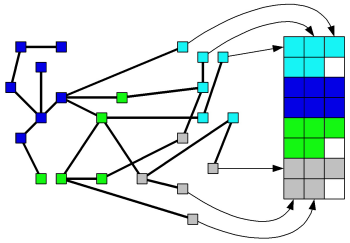

Fig. 4. Representing a graph on the GPU. Left: A graph spatially partitioned into partitions; right: a corresponding location texture

The location texture also holds the partition number of each node. Given a partition of maximum size c_{sz}, the height and width of each rectangular region representing a partition are set to $h_{partition} = max(8, \sqrt{C_{sz}})$ and $\lceil \frac{C_{sz}}{h_{partiton}} \rceil$, respectively.

Graph edges are represented by a *neighbors texture* and by an *adjacency texture*, as shown in Figure 5. The *adjacency* texture, whose size is $O(|E|)$, contains lists of (u,v) pointers into the location texture. These lists represent the neighbors of each node. The *neighbors* texture holds for each node a pointer into the adjacency texture, to the coordinates of the first neighbor of the node. Pointers to additional neighboring nodes are stored in consecutive locations in the adjacency texture. Doing so improves access locality. The degree of each node is also stored in the neighbors texture. Its size is equal to that of the location texture.

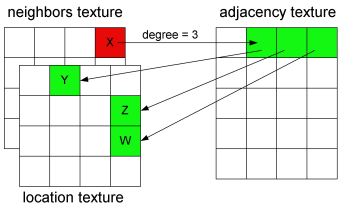

Fig. 5. Representing graph edges on the GPU. Node X has three neighbors: Y,Z and W.

The geometric (KD) partitions (described in Section 4, Step 6) are represented using two textures: the *partition information texture* and the *partition center of gravity texture*. The *partition information* texture holds the following information: (u0, v0) – the coordinates in the location texture of the upper left corner of the partition, the width and height of the partition rectangle in the location texture, the number of nodes in the last row of the partition (which may be partially filled), and the number of nodes in the partition. The *partition center of gravity* (C.G.) texture holds the current (x,y) coordinates of the center of gravity of each partition. Two textures are used to represent partitions not only because each texture is limited in the number of fields (to 4), but also to separate between the constant information and the information modified during the layout computation (i.e., center of gravity).

The forces computed during layout iteration are stored in two textures in a straightforward manner: the *attractive force texture* and the *repulsive force texture*. The *attractive force* texture contains for each node the sum of the attractive forces F^{attr} exerted on it by its neighbors. The *repulsive force* texture holds the sum of repulsive forces, F^{repl}: both by nodes in the same partition and by the other partitions in the graph. Both textures have the same dimensions as the location texture and contain the 2D components of the forces, (F_x, F_y).

Stream processing: On the GPU computation is performed by selecting the rendering target, which is the stream, or the texture, to which the output should be written. Next, an appropriate kernel program is loaded. Finally, graphics primitives such as quadrilaterals, are rendered in order to invoke the computation. For each pixel in the primitive (i.e., that the quadrilateral covers), the loaded kernel program is executed. Below we describe the order of invocations of the kernel programs, and their input and output textures. Figure 6 displays the *execution graph* of the algorithm.

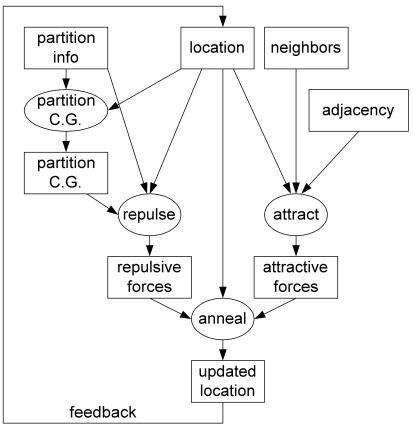

Fig. 6. Execution graph of GPU layout (rectangles = streams, ovals=kernels)

The algorithm is composed of three main stages, each implemented in a separate parallel_foreach loop which is executed in parallel for all elements on the GPU. The first loop calculates the center of gravity of each partition. The second loop calculates the forces acting on each node. The third loop displaces nodes using simulated annealing.

The *partition CG* (center of gravity) kernel calculates the center of gravity of each partition. The kernel reads information about each partition from the partition information texture and from the location texture and writes its result into the partition center of gravity texture. The GPU operates on all partitions in parallel.

The *repulse* kernel, which is the most time consuming kernel, calculates the repulsive forces exerted on each node. The kernel reads information from the partition information, the partition center of gravity, and the location textures. The output of the kernel is written to the repulsive force texture. For each fragment, the kernel first calculates the internal forces (exerted by nodes contained in the partition that the node belongs to). Then, it approximates the forces by all other partitions. Both of these calculations are performed using branching and looping instructions, in order to iterate over all other nodes in a partition and over all other partitions. Since the partitions are similarly sized, good branching consistency is maintained.

The *attract* kernel calculates the attractive forces caused by graph edges. It reads the neighbors, adjacency, and location textures and writes its output to the attractive forces texture. For each node, the kernel accesses the neighbors texture in order to get a pointer into the adjacency texture, which contains the (u,v) texture coordinates in the location texture, of the node's neighbors. For each neighboring node, the attractive force is calculated and accumulated.

Finally, the *anneal* kernel calculates the total force on each node. It reads the attractive force, repulsive force, and location textures and updates a second copy of the location texture. This double-buffering technique is used due to the inability of the GPU to read and write to the same stream. In the next iteration, the updated location texture is bound as input to the different kernels, thus facilitating *feedback* in our computation. The anneal kernel also bounds the total displacement of each node according to the current temperature of the layout. This

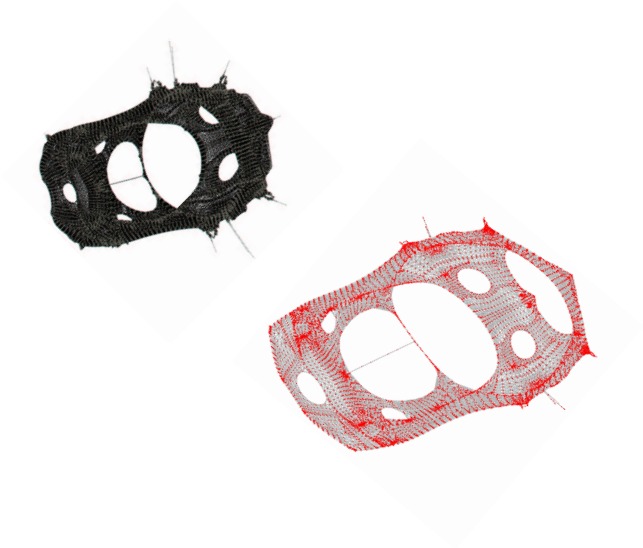

Fig. 7. bcsstk31. Red: our layout, black: FM^3 layout

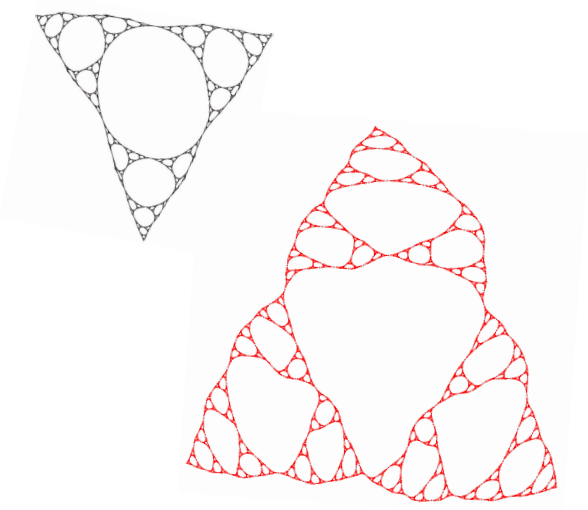

Fig. 8. Sierpinski_08. Red: our layout, black: FM^3 layout

temperature exponentially decreases at every iteration, hence allowing the graph to "freeze" into its final layout.

In total, the *partition CG* kernel performs $O(|V|)$ operations; the *repulse* kernel performs $O(|V|^{1.5})$ operations; the *attract* kernel performs $O(|E|)$ operations; and the *anneal* kernel $O(|V|)$ operations. On the GPU, the computations executed in each kernel, are run in parallel.

6 Results

Our algorithm was tested on several well-known graphs, commonly used in the graph drawing literature [39]. The bcsstk* graphs represent stiffness matrices. The Sierpinski graph is a self-similar fractal composed of triangles. The finan512 graph is taken from a linear programming matrix. The flower_B graph is constructed by joining 6 circles of length 50 at a single node before replacing each of the nodes by a complete subgraph with 30 nodes (K_{30}) [16]. The 4elt and crack graphs are 2D Finite–element meshes. The fe_* graphs are unstructured meshes related to fluid dynamics, structural mechanics, or combinatorial optimization problems. Figures 7 - 11 show some of the layouts computed by our algorithm, whereas Table 1 gives information about the graphs. Each image is accompanied with a layout

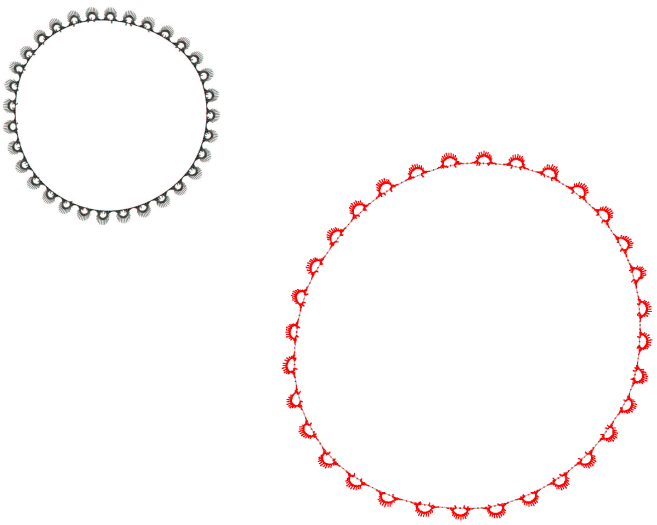

Fig. 9. finan512. Red: our layout, black: FM^3 layout

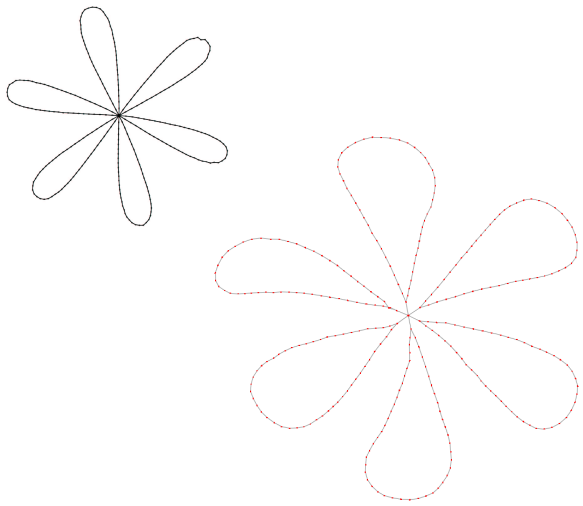

Fig. 10. flower_B. Red: our layout, black: FM^3 layout

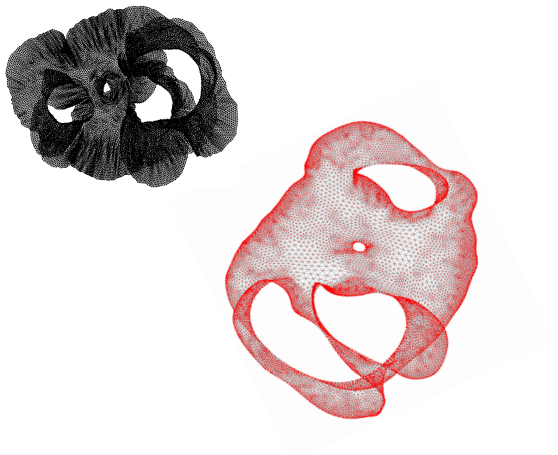

Fig. 11. 4elt. Red: our layout, black: Kamada-Kawai layout

graph	$\|V\|$	$\|E\|$	FM^3 algorithm 2.8GHz Pentium	our algorithm 3GHz Pentium	our algorithm 2.4GHz Core2 Duo	our algorithm 2.4GHz Core2 Duo + 8800GTS GPU
flower_B	9030	131241	11.9	3.25	2.21	1.59
4elt	14588	40176	$N\backslash A$	8.094	4.973	3.237
crack	10240	30380	23.0	4.844	3.018	2.44
bcsstk31	35586	572913	83.6	25.329	14.199	5.754
bcsstk32	44609	985046	110.9	39.266	22.549	9.617
bcsstk33	8738	291583	23.8	5.141	2.986	2.486
fe_pwt	36463	144794	69.0	22.985	13.48	5.44
finan512	74752	261120	158.2	79.268	43.645	12.267
fe_ocean	143437	409593	355.9	158.849	86.32	15.536
Sierpinski_08	9843	19683	16.8	5.25	3.127	2.705

Table 1. Graph information and running time [sec.]. Runtime columns show total running times for computing a layout.

computed by other algorithms [11, 16].

It can be seen that the layouts computed by our algorithm compare well with FM^3 [15]. The bcsstk31 graph (Figure 7) has a high edge density: $|E|/|V| = 16$. Moreover, it has a regular mesh-like structure. This regularity is extracted in our layout, as a result of the good partitioning and interpolation of the graph. Figure 8 shows the Sierpinski graph, which demonstrates that the symmetry of the graph is maintained, even though the holes in the graph are challenging, compared to more uniform mesh graphs. Figure 9 demonstrated the layout of the topologically challenging finan512. It is of similar quality to FM^3 and better than the other algorithms compared in [16]. Figure 10 shows the flower_B graph, which has a relatively high edge density: $|E|/|V| \geq 14$. Here, $k = 6$ is used for partitioning the graph and KK layout is performed on graphs up to 128 nodes. The 4elt graph, shown in Figure 11, exhibits large variations in node density and is thus challenging for an algorithm that seeks to maintain equal edge lengths [40]. The layout manages to show the interesting features of the graph – planarity and holes. Our layout is more uniform and contains less overlaps than the Kamada-Kawai layout from [11].

For the performance tests, a PC equipped with a 2.4 GHz Intel Core 2 Duo CPU and an NVIDIA 8800GTS GPU is used. Our algorithm was implemented in C++, Cg, and OpenGL. Table 1 shows the running time of our algorithm when using only the CPU and using the GPU to accelerate the computation. It also shows the running times for the FM^3 algorithm, produced on a 2.8 GHz Intel Pentium 4 CPU . In addition, it shows our algorithm on a slower machine (3.0 GHz Pentium 4), which is comparable to the machine used for the reported experiments of FM^3 [16].

Compared to FM^3, using a new GPU-equipped machine, a speedup by a factor of up to 22 times is achieved. The GPU accelerates the total computation time by a factor of up to 5.5. Without the GPU, on comparable hardware, our algorithm runs 2-4 times faster than FM^3.

7 Visualization of ISP Router Networks

We have applied our algorithm to the visualization of *Internet Service Provider (ISP)* router networks. The router networks of ISPs are comprised of several *points of presence (POPs)*. In each POP, several routers are located. They are connected to the backbone of the ISP and to routers connected to subscribers of the ISP. The data is taken from [1]. It was collected by using the traceroute tool to determine the route taken by packets traversing the ISP's network [36].

Figures 1, 12 show layouts of the networks of several ISPs. Each node in the graph corresponds to a router. Edges represent links between routers. Red nodes are not associated with any ISP in the data – they are used to connect the ISP to the rest of the Internet. The other nodes are color coded according to the ISP they belong to.

The layouts make evident some facts about these networks. First, most routers of each ISP are clustered together. This can be seen from the large clusters of nodes having the same color (excluding the red nodes). Second, two clusters are evident in Figure 12 – the brown cluster on the left, which represents an Australian ISP, and the rest of the graph. The yellow and pink nodes represent European ISPs. The black and blue nodes represent North American ISPs. The strongest

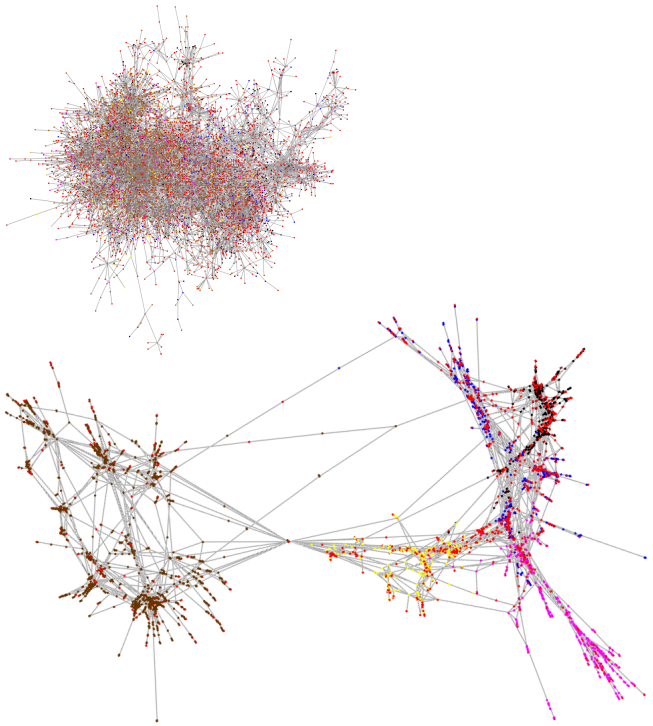

Fig. 12. ISP router map. Each node represents a router. Edges link routers. Red nodes are external to the ISPs visualized. Other nodes are colored according to the ISP they belong to: blue - Abovenet (US, 665 routers); black - Exodus (US, 554 routers); yellow - Ebone (Europe, 314 routers); pink - Tiscali (Europe, 514 routers); brown - Telstra (Australia, 3756 routers). A total of 10895 routers and 15667 connections are shown. Top left - GRIP layout. Bottom right - our layout.

connections exist between the two North American ISPs. There are good connections between European and North American ISPs. Connections between the Australian ISP and the other ISPs are sparser. Third, the per-ISP clusters are further divided into small clusters of routers, perhaps in the same city or nearby area. For instance, it can be seen that the brown routers belong to a couple of clusters. Fourth, the red external routers, which do not belong to any ISP, are used to link to the external world (outside the ISPs visualized). Fifth, the number of external routers is about the same as the number of internal routers, hence each router has one link on average to the world outside the ISP it belongs to. Sixth, the routers have varying degrees. Some have high degree and are central points (such as the router connecting the brown ISP and the yellow ISP), while others have low degree.

Figure 12 also compares our layout to one computed by GRIP [9]. It can be seen that GRIP's layout does not display the overall, clustered structure of the graph. Moreover, important edges, such as the ones connecting the brown cluster to the other part of the graph, are

not visible. However, the GRIP layout contains less overlap between nodes. To compare the performance, both layouts were computed using only the CPU on a 3GHz Pentium PC. Linux, required for GRIP, is not available on the PC with the GPU. The running time of GRIP was 3 seconds and the running time of our algorithm was 12 seconds. Trying to modify the parameters of GRIP resulted in a higher runtime, but without an improvement in layout quality.

8 Conclusion and Future Work

This paper has presented a new algorithm for multi-level force directed layout of graphs on the GPU. The algorithm has several key ideas. First, the graph is multi-level and is based on spectral partitioning. Second, the algorithm combines the strengths of both the Kamada–Kawai and Fruchterman– Reingold approaches, in order to compute a good layout fast. Third, a geometric partitioning and interpolation method in proposed, which facilitates the generation of good initial layouts of the finer versions of the graph.

Moreover, the paper has demonstrated how the GPU can be used to accelerate the algorithm by a factor of up to 5.5 times compared to our CPU implementation.

Last but not least, it has been demonstrated that the algorithm computes meaningful high quality layouts, while requiring significantly lower running times than existing algorithms of similar quality. Moreover, the algorithm was applied to visualize ISP networks.

In future research, we plan to implement more parts of the algorithm on the GPU. Acceleration candidates include the Fiedler vector computation and the initial position interpolation. Transitioning to the newly released NVIDIA CUDA development environment may help in getting finer control over the GPU.

Acknowledgements

This work was partially supported by European FP6 NoE grant 506766 (AIM@SHAPE) and by the Israeli Ministry of Science, Culture & Sports, grant 3-3421. Some images are courtesy of AT&T Research and the University of Köln. We thank them for allowing us to use them. We thank the reviewers for their helpful comments.

References

[1] Rocketfuel maps and data. `http://www.cs.washington.edu/research/networking/rocketfuel/` .

[2] J. Barnes and P. Hut. A hierarchical *O(N* log*N)* force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. on Graphics*, 23(3):777–786, 2004.

[4] F. R. K. Chung. Spectral graph theory. *Regional Conference Series in Mathematics, American Mathematical Society*, 92:1–212, 1997.

[5] R. Fernando, editor. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. 2004.

[6] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(100):619–633, 1975.

[7] Y. Frishman and A. Tal. Online dynamic graph drawing. In *EuroVis*, pages 75–82, 2007.

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

[9] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. *Comput. Geom*, 29(1):3–18, 2004.

[10] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *ACM / IEEE Supercomputing*, 2005.

[11] E. R. Gansner, Y. Koren, and S. C. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):457–468, 2005.

[12] J. Georgii, F. Echtler, and R. Westermann. Interactive simulation of deformable bodies on GPUs. In *SimVis*, pages 247–258, 2005.

[13] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 102–111, 2003.

[14] GPGPU. `http://www.gpgpu.org`.

[15] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing*, pages 285–295, 2004.

[16] S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Graph Drawing*, volume 3843 of *LNCS*, pages 235–250, 2005.

[17] C. D. Hansen, J. M. Kniss, A. E. Lefohn, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.

[18] D. Harel and Y. Koren. A Fast Multi-Scale Algorithm for Drawing Large Graphs. *J. Graph Algorithms Appl.*, 6(3):179–202, 2002.

[19] D. Harel and Y. Koren. Drawing graphs with non-uniform vertices. In *Proc. Working Conference on Advanced Visual Interfaces (AVI'02)*, pages 157–166. ACM Press, 2002.

[20] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. *J. Graph Algorithms Appl*, 8(2):195–214, 2004.

[21] M. J. Harris, W. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 92–101, 2003.

[22] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier volume rendering on the GPU using a split-stream-FFT. In *Vision, modeling and visualization*, pages 395–403, 2004.

[23] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[24] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*. 2001.

[25] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based particle engine. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 115–122, 2004.

[26] Y. Koren, L. Carmel, and D. Harel. Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling & Simulation*, 1(4):645–673, 2003.

[27] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *Proc. ACM SIGGRAPH*, volume 22(3) of *ACM Transactions on Graphics*, pages 908–916, 2003.

[28] Y. Liu, X. Liu, and E. Wu. Real-time 3D fluid simulation on GPU with complex obstacles. In *Pacific Conference on Computer Graphics and Applications*, pages 247–256, 2004.

[29] K. Moreland and E. Angel. The FFT on a GPU. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 112–119, 2003.

[30] L. Nyland, M. Harris, and J. Prins. The rapid evaluation of potential fields using programmable graphics hardware. In *ACM Workshop on General Purpose Computing on Graphics Hardware*, 2004.

[31] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics*, pages 21–51, 2005.

[32] M. Pharr and R. Fernando, editors. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. 2005.

[33] Pothen, A., Simon, H., and Liou, K. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. and Appl.*, 11:430–452, 1990.

[34] A. J. Quigley and P. Eades. FADE: Graph drawing, clustering, and visual abstraction. In *Graph Drawing*, number 1984 in LNCS, pages 197–210, 2000.

[35] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. on PAMI*, 22(8):888–905, 2000.

[36] N. T. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with rocketfuel. In *SIGCOMM*, pages 133–145, 2002.

[37] E. Tejada and T. Ertl. Large Steps in GPU-based Deformable Bodies Simulation. *Simulation Modelling Practice and Theory*, 13:703–715, 2005.

[38] I. G. Tollis, G. D. Battista, P. Eades, and R. Tamassia. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[39] C. Walshaw. graph collection. `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.

[40] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.

[41] D. S. Watkins. *Fundamentals of Matrix Computations*. John Wiley, 2002.