Rapid Multipole Graph Drawing on the GPU

Apeksha Godiyal[1], Jared Hoberock[1], Michael Garland[2], and John C. Hart[1]

[1] University of Illinois
{godiyal2,hoberock,jch}@illinois.edu
[2] NVIDIA Corp.,
mgarland@nvidia.com

Abstract. As graphics processors become powerful, ubiquitous and easier to program, they have also become more amenable to general purpose high-performance computing, including the computationally expensive task of drawing large graphs. This paper describes a new parallel analysis of the multipole method of graph drawing to support its efficient GPU implementation. We use a variation of the Fast Multipole Method to estimate the long distance repulsive forces in force directed layout. We support these multipole computations efficiently with a k-d tree constructed and traversed on the GPU. The algorithm achieves impressive speedup over previous CPU and GPU methods, drawing graphs with hundreds of thousands of vertices within a few seconds via CUDA on an NVIDIA GeForce 8800 GTX.

1 Introduction

Automatic graph layout algorithms convert the topology of vertex adjacency into the geometry of vertex position. These layouts usually represent vertices as points or icons in two or three dimensions connected by edges represented by lines or arcs. Automatic graph drawing has many important applications in information visualization, software engineering, database, web design, networking, VLSI circuit design, social network analysis, cartography, bioinformatics and the organization of visual interfaces for many other domains [4]. Growth in information technology and data processing has increased the size and complexity of graph datasets, posing the problem of drawing large graphs with millions of nodes that demand the consideration of new scalable parallel approaches.

Classical force directed algorithms [7, 9, 12, 22] layout graphs of hundreds of vertices, but run in $O(|V|^2 + |E|)$ time and do not scale well for larger graphs. Approximate force directed techniques [13, 14, 18, 20, 32] perform better, using a multilevel approach based on a graph hierarchy, where smaller coarser graph levels guide the initial drawing of progressively larger, finer levels of the graph hierarchy. The class of algorithms based on linear algebra [21, 23] are even faster. They perform best on grid-like regular graphs but can condense features on other graph types (e.g. with many biconnected components) [19, 21, 23].

These state-of-the-art algorithms for straight line graph drawing can still run too slow on modern graphs, e.g. six minutes for a graph of 143,437 nodes [18]. Other approaches work efficiently but with uneven layout quality across graph type, e.g. extremely fast ACE[23] and HDE[21] methods work best only on quasi-grids. To address both limitations, this paper reworks the general-graph quality of approximated force directed layout into a form that can be efficiently processed on the GPU to layout hundreds of thousands of nodes within a few seconds. Our GPU implementation of the fast multipole multilevel method (FM^3) is more than 20$\times$ faster than the latest reported CPU version [18].

We parallelize a potential field based multilevel algorithm that uses only multipole expansions (no local expansions) to approximate long distance forces. This combines Barnes-Hut [3] and fast multipole methods (FMM) [16]. The FMM approach has proven error bounds and better asymptotic complexity, whereas Barnes-Hut is popular due to its simplicity and a low associated constant factor of implementation [15]. Their hybrid enjoys good error bounds and an $O(|V| \log |V| + E)$ time complexity with low constant factor, and yeilds high quality layouts that represent both local and global structures well, even for graphs deemed challenging [19].

The modern graphics processing unit (GPU) was initially designed for raster-based videogame graphics, but its marked improvement in performance and programmability has generated considerable interest in it as a high-performance computing platform [27, 29]. However, GPU programming remains challenging, and its performance relies on the ability to decompose a task into concurrent identical data-parallel instruction threads with limited support for stacks or recursion, and managing their access patterns to the various kinds of memory (shared, local, CPU, etc.). The contributions of this paper are the systems-level design and deployment of an efficient manycore graph drawing algorithm and to show that the acceleration of multipole-based layout justifies the challenges posed by the GPU's architecture and programming.

The main challenge of FMM processing on a single-instruction multiple-data (SIMD) processor (such as a GPU) is managing a shared spatial hierarchy. The k-d tree has been a popular choice for particle simulation [8, 2] as its size complexity is distribution independent [31], but does not map easily to the GPU's SIMD programming model. We combine the CPU and GPU to construct the tree, using the GPU for fast median selection so the CPU can construct a balanced k-d tree with $O(\log N)$ depth that keeps force calculation within $O(N \log N)$. We traverse the structure entirely on the GPU, using an efficient "stackless" k-d tree representation, where each node has a pair of pointers, one pointing to the first child and the other to the next node (in pre-order traversal order). Each processor of a data-parallel SIMD processor can efficiently traverse such a hierarchy by simply following one of two pointers [6, 10].

2 Related Work

The Fast Multipole Multilevel Method (FM^3) produces pleasing layouts in the general case and is relatively fast [18]. It combines a multilevel spatial partitioning with a multipole approximation of all pairs repulsive forces, specifically Greengard's FMM algorithm [16]. Our new GPU version uses only the multipole expansion coefficients and not the local expansion coefficients to approximating repulsive forces. We show that these multipole expansion coefficients alone are sufficient to produce high quality layout and the added complexity of working with local expansion coefficients is unnecessary. Our GPU implementation is $20\times - 60\times$ faster than the preveious CPU implementations of FM^3. Another improvement over the previous CPU FM^3 implementation [18] is that we use a k-d tree instead of quad tree for force calculations, motivated by GPU architecture as elaborated in Sec. 4.1.

Our implementation is more than 30% faster than a previous GPU multilevel force directed graph layout method [11]. That method approximated the all-pairs repulsive force with a center of gravity multipole acceptance criteria, which when compared to FM^3 has a larger aggregate error that can even become unbounded for unstructured distributions [28]. Our approach's time complexity, $O(|V|\log|V| + |E|)$, improves their's, $O(|V|^{1.5} + |E|)$.

Others have implemented general-purpose FMM on the GPU [30, 17]. Their approaches differ from ours as they include all FMM steps, most of which are unnecessary for graph drawing. Our approach utilizes the k-d tree which outperforms their quadtree, and we focus specifically on the issue of GPU tree construction.

3 Algorithm

Multilevel layout methods significantly reduce running times by converging to the optimal layout in fewer iterations [18, 23, 20, 14, 13, 32]. This approach recursively coarsens an input graph G^0 to produce a series of smaller graphs $G^1 \dots G^k$, until the size of the coarsened graph falls below a threshold. An initial layout is first computed iteratively for the coarsest graph G^k. The converged vertex positions of a level i graph G^i are used as the initial vertex positions of the next finer level $i-1$ graph G^{i-1}, which should relax into a converged state after a few iterations. This continues until the layout for the finest graph (the input), G^0, is obtained.

We use the multilevel method shown in Algorithm 1. The *ComputeLayout* step is the most expensive with runtime complexity of $O(|V|\log|V| + |E|)$, and is accelerated by the GPU. The remaining functions are linear $O(|V|)$ and computed on the CPU.

3.1 Coarsening

The function *CoarsenGraph* coarsens by maximal independent set (MIS) filtration, which has the advantage of being simple, efficient and produces a filtration controlled by the geometry of the graph [14, 13]. The vertex subset $S \subset V$ is an independent set of a graph $G = (V, E)$ if no two elements of S are connected by an edge. A maximal independent set filtration of G is a family of sets $V = V^0 \supset V^1 \supset \dots \supset V^k \supset \emptyset$, such that each V^i is an independent set of V^{i-1}.

Calculating optimal independent sets is a NP-Complete problem, though an efficient 2-approximation exists. An independent set S of a set V can be computed by repeatedly deleting a vertex $v \in V$ and adding it to S and removing all vertices adjacent to v from V, until V is empty. The set S is the desired independent set.

Algorithm 1. Overall Algorithm

```
Input: $G = (V, E)$ with random initial placements
Output: $G = (V', E)$ with final placements
initialization;
graph $G^0 \longleftarrow G$;
$threshold \longleftarrow 50$;
$i \longleftarrow 0$;
while $|V^i| \geq threshold$ do
    graph $G^{i+1} \longleftarrow CoarsenGraph(G^i)$;
    $i \longleftarrow i + 1$
end
while $i \geq 0$ do
    ComputeLayout($G^i$) ; /* via the GPU */
    if $i \geq 1$ then
        InterpolateInitialPositions($G^{i-1}$)
    end
    $i \longleftarrow i - 1$
end
return $G^0$
```

3.2 Interpolation

The function *InterpolateInitialPositions* derives the starting positions of vertices in G^i from the positions of vertices in the converged layout of G^{i+1}, using a relaxation method [11]. Each vertex $v \in V^i$ is initially placed at the position of its parent vertex $v' \in V^{i+1}$. Then several iterations (we used a maximum of 50) of a form of graph Laplacian move each vertex to an average of its current position, p_i, and that of its neighbors N_i,

$$p_i = \frac{1}{2}\left(p_i + \frac{1}{\deg(i)} \sum_{j \in N_i} p_j\right). \quad (1)$$

3.3 Force Calculation

For each graph G^i, the function *ComputeLayout* iteratively calculates and applies forces until it converges. The coarsest graph G^k typically requires 300 iterations, but this number decays rapidly for finer graphs and in most cases the finest graph G^0 needs zero iterations to converge. The pseudocode for one iteration is given in Algorithm 2.

3.4 Force Model

As in the force directed algorithm [12], we assume that the vertices of a graph $G(V, E)$ are charged particles that repel each other with an inverse-square law, and the edges are springs that contract with a non-physical but effective force [18]

Algorithm 2. Force Calculation Algorithm

```
Input: $G = (V, E)$ with initial placements
Output: $G = (V', E)$ with final placements
$kdTree \longleftarrow constructKDTree(V)$
Spawn $|V|$ threads on the GPU ; /* Thread i calculates force on $v_i$ */
foreach thread i do
    $force \longleftarrow calculateRepulsion(v_i, kdTree)$
    $force \longleftarrow force + calculateAttraction(v_i, E)$
    Send calculated force values to CPU in an array
end
; /* Done on the CPU to avoid global synchronization on the GPU */
forall $v_i$ do
    moveVertex($v_i$, force)
end
return $G$
```

$$F = d^2 \log(d/d') \tag{2}$$

where d and d' are the actual and desired lengths of the edge.

3.5 Multipole Calculation

The most expensive step in force directed graph drawing is the all-pair repulsive force calculations. Although the force calculations may be quite complex in the near-field (when two vertices are very close to each other), force calculations are well-behaved in the far-field. In particular, if a vertex is sufficiently far from a set of charges, we may compute the aggregate effect of the charges on that vertex, and need not resort to computing every interaction. Greengard [16] first demonstrated how potential field based approximations can be used to find the far-field forces using quad trees. The idea is to construct a tree based spatial partition of particles and then evaluate multipole expansions using this tree.

Theorem 1. *(Multipole Expansion)Suppose that m charges of strengths q_i are located at points z_i, for $i = 1 \ldots m$, with center z_0 and $|z_i - z_0| < r$. Then for any $z \in C$ with $|z - z_0| > r$, the potential $\Phi(z)$ induced by the charges is given by*

$$\Phi(z) = Q \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k} \tag{3}$$

where $Q = \sum q_i$ and $a_k = \sum -q_i(z_i - z_0)^k/k$. As force is the negative of the gradient of the potential, the force that acts on a particle of unit charge at position z is given by $(\mathrm{Re}(\Phi'(z)), -\mathrm{Im}(\Phi'(z)))$.

Instead of summing up an infinite series for (3), only a constant number p of terms are calculated. The resulting truncated Laurent series is called *p-term multipole expansion.* We choose $p = 4$ as it is sufficient to keep the error of the approximation less than 10^{-2} [18].

As the k-d tree is constructed, the coefficients of this multipole expansion are calculated and stored for each node using (3). The center of a k-d tree node is the geometric center of the rectangular region it represents, and the radius used is the radius of a circle circumscribing this rectangular region. Each node in the k-d tree thus maintains a collection of charges (vertices of the graph) lying in its rectangular regions. Let $G(V, E)$ be a graph and K be the k-d tree of the vertices of G. Let n be a node of K with center z_0 and radius r. Let $\{v_i, v_2, \ldots, v_k\}$ be the set of vertices of graph G that are contained in k-d tree node n. To calculate the approximate repulsive force on each vertex $v \in V$ located at z, K is traversed from the root node. At a node n, if the distance between z_0 and z is greater than r, then the approximate repulsive force between v and vertices $v_i\{i = 1, \ldots, k\}$ are calculated using (3). Otherwise, if n is an internal node, the process is repeated for its children, and if n is a leaf node, the exact repulsive forces are calculated.

4 GPU Implementation

4.1 Processing the K-D Tree

Unlike the more traditional quadtree used in n-body simulation, we used a k-d tree [5]. Aluru *et al.*[1] has shown that the running time of adaptive FMM using quad tree [16] depends on the particle distribution and cannot be bounded in number of particles. In order to remedy this and guarantee $O(|V|log|V|)$ running time complexity, [18] uses complicated tree thinning and balancing techniques. These techniques do not translate into efficient GPU implementation because of the lack of recursion (no unbounded stack) and dynamic memory allocation. Since the k-d tree is a density decomposition tree and not a spatial decomposition tree, it does not suffer from distribution dependent running time [31].

The CUDA GPU programming model has a complex memory hierarchy and one has to keep in mind multiple factors to achieve good performance [26]. The k-d tree is traversed by all of the GPU threads and all the threads need the vertex position data for near field and attractive force calculations. Thus these data structures are passed to the GPU in texture memory, which is cached yielding higher bandwidth from k-d tree node locality. In our implementation, the k-d tree is constructed for the first four iterations and then for every twentieth iteration, because it changes only slightly in each later iteration and these changes do not significantly impact force calculations.

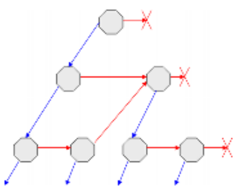

Fig. 1. A "stackless" k-d tree pre-threaded with first child (blue) and next neighbor (red) pointers

Traversal. Stackless traversal of the k-d tree on the GPU is achieved by a structure shown in Fig. 1 Each node of the tree has two pointers. The blue (success) pointer indicates its first child whereas the red (failure) pointer points to its next neighboring node. This tree threading allows the streaming SIMD GPU processing to parse a hierarchical data structure efficiently [6, 10]. The data parallel SIMD architecture of the GPU requires that when control flow reaches a condition, if some processors follow one side of the condition and the rest of the processors follow the other side of the condition, then all of the processors need to evaluate both sides of the condition, zeroing out the result of the side not used by each processor. Tree threading allows the processors instead to simply follow one of two pointers, replacing conditional control flow with data indirection which is fully supported by the GPU.

Construction. A k-d tree is constructed recursively. Each node of a k-d tree divides the set of vertices it represents V, into two equal sets by splitting along a chosen dimension. (In our implementation, the splitting dimension alternates between the two axes.) This bisection is achieved by a radix selection algorithm [24] whose worst case time complexity is $O(|V|)$. The process of finding the median and splitting the set of vertices is applied recursively until a node has less than threshold number of vertices (four, in our implementation). The multipole expansion coefficients from (3) of each node are calculated as the k-d tree is constructed. This median splitting approach generates a balanced k-d tree in $O(|V| \log |V|)$ time.

The radix selection algorithm is faster on the GPU for arrays of large size. In our configuration, the crossover array size, for which the GPU radix selection is faster than a well tuned CPU implementation, is 50,000, and we use the CPU for smaller arrays. We implemented radix selection using efficient GPU scan primitives [29] (which have also been used for GPU radix sort [25]).

4.2 Radix Selection with Prefix Scan

Radix select is the selection analog of the radix sort algorithm. It is recursive and selects the key (vertex coordinate in our case) whose rank is m, from an array $A[1 \dots n]$ of n keys. The array is split at position s, into two sub-arrays based on the most significant bit: $A[1 \dots s]$ contains all keys with 0 as the most significant bit, and $A[s+1 \dots n]$ contains all keys with 1 as the most significant bit. Then the next significant bit is considered. This goes on recursively until the key with rank m is found.

To carry out the split at each level of recursion in parallel, each thread needs to copy a different input key $A[i]$ to the split array. The address of each key $A[i]$, is the number of keys in $A[1 \dots i-1]$ whose most significant bit is 0. The array of these counts is called the prefix sum of A, denoted here as $B[1 \dots n]$ such that $B[i] = \sum_{j<i} A[j]$. We compute this prefix sum on the GPU using an efficient O(n) CUDA prefix scan implementation [29]. This work-efficient scan of n elements requires two passes over the array: *reduce* and *down-sweep*. Each

requires $\log(n)$ parallel steps. The amount of work is cut in half at each step, resulting in an overall work complexity of $O(n)$.

4.3 Compressed Sparse Row Representation

We use a compressed sparse row (CSR) format, essentially a sparse matrix data structure [29], for representing the edges of the graph in GPU texture memory. It avoids conditional statements and thus makes the implementation fast. Let i be a vertex of graph G such that i has k edges $(i, j_1), (i, j_2)...(i, j_k)$. Then the graphs adjacency list is represented by 2 arrays:

1. Edge-value: For each vertex i, this array stores vertices $\{j_1, j_2...j_k\}$ i.e. the adjacency list of i.
2. Edge-index: Edge-Index[i-1] and Edge-Index[i] store the beginning and ending of the adjacency list of vertex i.

For each vertex i, a GPU processing thread uses this CSR representation to calculate the attractive forces due to its incident edges. This parallel computation is not perfectly load-balanced as the work done by each thread depends on the degree of the vertex it is handling. Processing the edges instead of the vertices would rectify this, but would require either atomic operations for adding up all the forces on a single vertex, or a prefix sum to add up the forces calculated by different threads, and neither option is very efficient.

The *edge-value* array is accessed frequently by each thread, and so is placed in the cached texture memory of the GPU. The *edge-index* array is accessed only twice per thread with negligible gain from caching, and so is placed in plain read-write GPU memory.

5 Results

The algorithm was tested on a single core 2.21 GHz AMD Athlon(tm) 64 Processor running Windows XP, with an NVIDIA GeForce 8800 GTX card programmed via the CUDA (Compute Unified Device Architecture) programming model, compiled by a C compiler with language extensions [26]. Both CPU and GPU implementations used single precision floating point.

The algorithm was tested on a variety of graphs extensively used in graph drawing research to support comparisons [18, 19, 33]. Figure 2 shows selected layouts and their associated run times. The layouts of all the tested artificial and real-world graphs resemble those produced by FM^3 [18]. Like FM^3, our algorithm is able to display the regularity of six-ary trees, the symmetry of spider and flower graphs and the global structure of snowflake graphs.

Figure 3 shows for various graphs the speedup our implementation achieves over FM^3 and over the GFDL force directed layout GPU implementation [11]. It shows our implementation to be $1.3\times - 4\times$ faster than GFDL and $20\times - 60\times$ faster than CPU implementation of FM^3. Figure 4 demonstrates the scalability of our GPU implementation. Its running time is largely a factor of graph size, though dependent

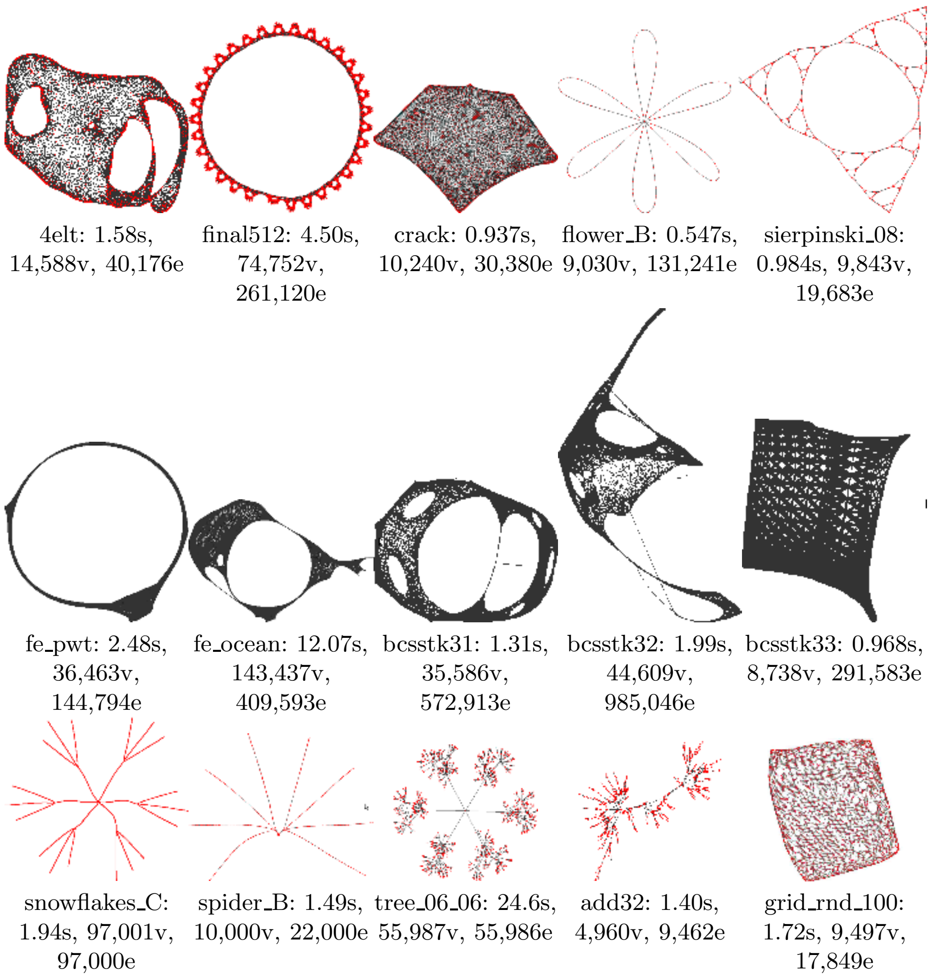
4elt: 1.58s, 14,588v, 40,176e
final512: 4.50s, 74,752v, 261,120e
crack: 0.937s, 10,240v, 30,380e
flower_B: 0.547s, 9,030v, 131,241e
sierpinski_08: 0.984s, 9,843v, 19,683e
fe_pwt: 2.48s, 36,463v, 144,794e
fe_ocean: 12.07s, 143,437v, 409,593e
bcsstk31: 1.31s, 35,586v, 572,913e
bcsstk32: 1.99s, 44,609v, 985,046e
bcsstk33: 0.968s, 8,738v, 291,583e
snowflakes_C: 1.94s, 97,001v, 97,000e
spider_B: 1.49s, 10,000v, 22,000e
tree_06_06: 24.6s, 55,987v, 55,986e
add32: 1.40s, 4,960v, 9,462e
grid_rnd_100: 1.72s, 9,497v, 17,849e

Fig. 2. Layouts of various graphs computed with out approach, indicated by name, running time (in seconds), followed by the numbers of vertices and edges

on the number of iterations needed to resolve vertex placement at each level of the graph hierarchy. Thus the large 6-ary tree required significantly more iterations (by a factor of five) to reach a planar embedding than did the others.

We recorded the running time of the major parts of the algorithm for both the CPU and the GPU implementations. Table 1 shows the result for a few graphs. The CPU implementation spends on an average nearly 85.5% of CPU cycles in calculating the forces and this step is clearly the performance bottleneck. The GPU implementation reduces the time spent in calculating forces by 7-40 times (depending upon the size of the graph). One disadvantage of the GPU implementation is that lots of cycles are wasted in copying data back and forth between the GPU and the CPU. GPU implementation spends 18%-25% of the

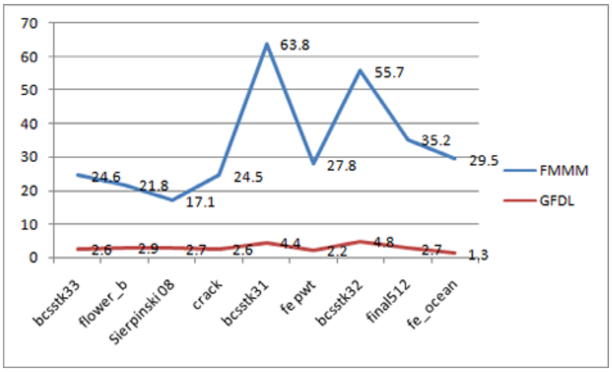

Fig. 3. Speedup factors over GPU force directed layout (GFDL) and Fast Multilevel Multipole Method (FMMM). The graphs are in increasing order of graph size.

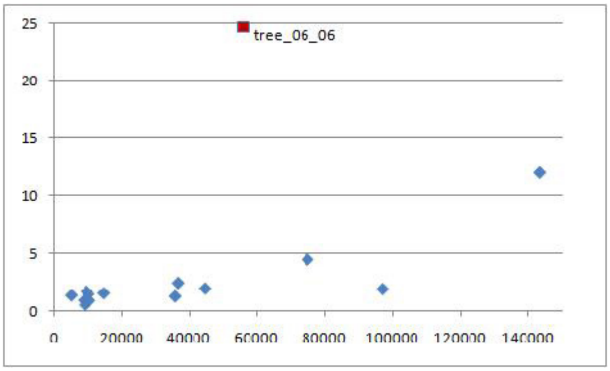

Fig. 4. Running time vs. graph size for GPU accelerated FM^3 layout

Table 1. Running time (in seconds) comparing total and component run times on CPU (numerator) v. GPU (denominator)

Graph	$\|V\|$	$\|E\|$	Total	Coarsening	Data Trans.	Tree Const.	Force Calc.
bcsstk33	8,738	291,583	1.63 / 0.968	0.0 / 0.0	0.032 / 0.141	0.095 / 0.096	1.48 / 0.242
4elt	14,588	40,176	7.23 / 1.58	0.0 / 0.0	0.172 / 0.375	0.516 / 0.375	5.92 / 0.672
crack	10,240	30,380	3.51 / 0.937	0.0 / 0.0	0.080 / 0.172	0.456 / 0.203	2.81 / 0.449
final512	74,752	261,120	81.55 / 4.50	0.25 / 0.25	0.260 / 0.828	3.39 / 1.49	73.8 / 1.932
fe_ocean	143,437	409,593	90.9 / 12.07	4.1 / 4.1	1.30 / 1.50	5.20 / 3.89	83.0 / 2.48

running time in data movement as compared to 2%-3% time spent by the CPU implementation on the same. Time for constructing the k-d tree is nearly same in the CPU and GPU implementations, for graphs with less than 50,000 vertices. For larger graphs, k-d tree construction is more than 30% faster on the GPU.

6 Conclusions and Future Work

The parallel algorithm described in this paper makes graph drawing significantly faster without compromising layout quality, improving previous fast implementations that were limited to grid-like graphs. The speedup obtained shows that it is now possible to draw general graphs with hundreds of thousands of nodes within a few seconds via the GPU. We also showed that for the purpose of graph drawing multipole expansions suffice, and local expansions in FMM should be best avoided due to their the high constant factor.

The optimized layout of each graph required the hand tuning of a number of parameters, as automatic inference of these optimal parameters remains an open research problem. Further algorithm improvements may be possible. Increasing CPU-GPU bandwidth may lower the 50,000-node limit where the GPU outpaced the CPU on median finding, and further load balancing may improve force calculation.

Acknowledgments

This work is supported by the NSF under the grant #0534485, and by NVIDIA Corp.

References

[1] Aluru, S., Prabhu, G.M., Gustafson, J.: Truly distribution-independent algorithms for the n-body problem. In: Proc. Supercomputing, pp. 420–428 (1994)
[2] Appel, A.W.: An efficient program for many-body simulation. SIAM J. Sci. & Stat. Comp. 6(1), 85–103 (1985)
[3] Barnes, J., Hut, P.: A hierarchical o(n log n) force-calculation algorithm. Nature 324(6096), 446–449 (1986)
[4] Batini, C.: Applications of graph drawing to software engineering (abstract). SIGACT News 24(1), 57 (1993)
[5] Bentley, J.L.: Multidimensional binary search trees used for associative searching. CACM 18(9), 509–517 (1975)
[6] Carr, N.A., Hoberock, J., Crane, K., Hart, J.C.: Fast gpu ray tracing of dynamic meshes using geometry images. In: Proc. Graphics Interface, pp. 203–209 (2006)
[7] Davidson, R., Harel, D.: Drawing graphs nicely using simulated annealing. ACM Trans. Graph. 15(4), 301–331 (1996)
[8] Dikaiakos, M.D., Stadel, J.: A performance study of cosmological simulations on message-passing and shared-memory multiprocessors. In: Intl. Conf. on Supercomputing, pp. 94–101 (1996)
[9] Eades, P.A.: A heuristic for graph drawing. Congressus Numerantium 42, 149–160 (1984)
[10] Foley, T., Sugerman, J.: Kd-tree acceleration structures for a GPU raytracer. In: Proc. Graphics Hardware, pp. 15–22 (2005)
[11] Frishman, Y., Tal, M.-A.: Multi-level graph layout on the gpu. IEEE Trans. Vis. Comp. Graph. 13(6), 1310–1319 (2007)

[12] Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Software - Practice and Experience 21(11), 1129–1164 (1991)
[13] Gajer, P., Goodrich, M.T., Kobourov, S.G.: A multi-dimensional approach to force-directed layouts of large graphs. Comput. Geom. Theory Appl. 29(1), 3–18 (2004)
[14] Gajer, P., Kobourov, S.G.: GRIP: Graph dRawing with intelligent placement. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 222–228. Springer, Heidelberg (2001)
[15] Grama, A.Y., Kumar, V., Sameh, A.: Scalable parallel formulations of the Barnes-Hut method for n-body simulations. In: Proc. Supercomputing, pp. 439–448 (1994)
[16] Greengard, L.F.: The rapid evaluation of potential fields in particle systems. Ph.D. thesis, Yale, New Haven, CT, USA (1987)
[17] Gumerov, N.A., Duraiswami, R.: Fast multipole methods on graphics processors. J. Comp. Physics 227, 8290–8313 (2008)
[18] Hachul, S., Jünger, M.: Large-graph layout with the fast multipole multilevel method. Tech. rep., Zentrum für Angewandte Informatik Köln (December 2005)
[19] Hachul, S., Junger, M.: An experimental comparison of fast algorithms for drawing general large graphs. In: Healy, P., Nikolov, N.S. (eds.) GD 2005. LNCS, vol. 3843, pp. 235–250. Springer, Heidelberg (2006)
[20] Harel, D., Koren, Y.: A fast multi-scale method for drawing large graphs. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 183–196. Springer, Heidelberg (2001)
[21] Harel, D., Koren, Y.: Graph drawing by high dimensional embedding. In: Goodrich, M.T., Kobourov, S.G. (eds.) GD 2002. LNCS, vol. 2528. Springer, Heidelberg (2002)
[22] Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. Inf. Process. Lett. 31(1), 7–15 (1989)
[23] Koren, Y., Carmel, L., Harel, D.: ACE: a fast multiscale eigenvectors computation for drawing huge graphs (2001)
[24] Mahmoud, H.M.: Sorting: A Distribution Theory, chap. High Qulaity Ambient Occlusion. Wiley Interscience, Hoboken (2000)
[25] NVIDIA: CUDA data parallel primitives library, `http://www.gpgpu.org/developer/cudpp/`
[26] NVIDIA: CUDA programming guide (2007), `http://developer.nvidia.com/object/cuda.html`
[27] Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Reading (2005)
[28] Sarin, V.: Analyzing the error bounds of multipole-based treecodes. In: Proc. Supercomputing, p. 19 (1998)
[29] Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Proc. Graphics Hardware, August 2007, pp. 97–106 (2007)
[30] Stock, M.J., Gharakhani, A.: Toward efficient gpu-accelerated n-body simulations. In: 46th AIAA Aerospace Sciences Meeting & Exhibit (2008)
[31] Uhlmann, J.K.: Enhancing multidimensional tree structures by using a bi-linear decomposition. Natl. Tech. Info. Svc. ADA229756 (1990)
[32] Walshaw, C.: A multilevel algorithm for force-directed graph drawing. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 171–182. Springer, Heidelberg (2001)
[33] Walshaw, C.: Graph collection (2007), `staffweb.cms.gre.ac.uk/~wc06/partition/`