ECE1724H S Lecture 102:
Special Topics in
Programming Massively Parallel Graphics Multiprocessors Using CUDA

Force Directed Placement: GPU implementation

Bernice Chan, 990338951
Ivan So, 991731854
Mark Teper, 991603886

1. Introduction / Motivation

Automated graph drawing remains a difficult placement and layout problem. This problem is difficult in part due to the complexity of formulating good algorithms to draw graphs which are aesthetically pleasing for human visualization. Graph layout algorithms use a heuristic to determine if they are getting closer or farther from being aesthetically pleasing and use an iterative method to progress towards the final graph layout solution.

The graphing problem consists of a number of *elements* (or graph nodes) which are interconnected in some manner. The connections between elements are referred to as *edges***.** For example, in a family tree – each person would be an element and parents would be connected to their children through edges. The goal of the algorithm is to take this data and present it to the user in an easy to understand way by generating a graphical representation.

Applications of graph drawing include VLSI circuit design, network relationship analysis, cartography, and bioinformatics [1]. The layout algorithm can also be used for a large class of applications: from drawing family-trees to laying out class relationship diagrams in software systems.

We propose to develop a force-directed placement algorithm that will solve the graph layout problem by using an energy minimization technique. In this approach, the aesthetic quality of the diagram is mapped to an energy state, with the algorithm attempting to search for a minimum.

This work aims to demonstrate the performance advantage of a GPU implementation (using NVIDIA CUDA) of a force-directed graph layout placement algorithm as compared to a CPU implementation. Our GPU implementation takes advantage of the inherent parallel computational power available in the GPU and with some additional memory bandwidth optimizations for our algorithm, the GPU implementation was able to achieve up to a 58x speed-up as compared to the CPU.

2. Related Work

Improved heuristics for graph layout have been developed for both force directed placement and simulated annealing approaches. Force directed placement algorithms for graph layout are primarily based on Hooke's law, but have deviated from strict physical modelling to better match the requirements of different graphing applications as described by Eades spring-mass equations [2] and later adapted by Fruchterman and Reingold to emulate particle physics in a simulated annealing algorithm [3].

Graph placement and layout remains an active area of research, with Frishman and Tal [4] [5] using GPUs to speed-up incremental graph layout to provide results 8x faster than a CPU, and proposing different approaches to partition the computational work to improve parallelism.

For our work, we chose to use the simpler heuristics proposed by Eades which lend themselves well to parallelization on the GPU. The focus of our work was to demonstrate that a significant runtime advantage could be achieved on a GPU even for a simple force directed placement algorithm by increasing parallelism and optimizing memory bandwidth utilization.

3. Algorithm

Two graph placement & layout algorithms were considered: Simulated Annealing and Force-directed placement algorithms

Simulated Annealing

In the simulated annealing model, element moves are performed randomly while searching for a minimum energy state. The algorithm evaluates each random move by computing a change in the "cost function" or "energy state" of the system. If the randomly selected move reduces the energy state or at worst increases by less then a threshold temperature factor "T" then the move is accepted, otherwise it is undone and another move is performed. Over time the temperature factor decreases, allowing only better and better moves to be accepted. This decreasing of the threshold temperature allows the algorithm to "escape" local minima in the early stages of evaluation, while allowing the algorithm to converge on a good solution. The energy-state (or cost function) in the simulated annealing model can use the same model as the one used by the force-based placement algorithm.

Force-based/Force-directed algorithms

In this model, each element in the graph is modelled as a mass, with edges between graph elements modelled as springs as shown in Figure 1. Values for the "gravitational force" and "spring constant" can be varied to produce different results. Further, additional forces (electrical charges, logarithmic springs, moments, damping effects) can be used to alter the aesthetic. The algorithm takes this model and chooses some arbitrary initial state. It then iteratively simulates the movement of each element (or "mass") by through increments of time (steps) until the model reaches steady state. On each iteration, the forces acting on each element in the model are computed and the element's "position" and "velocity" are then adjusted.

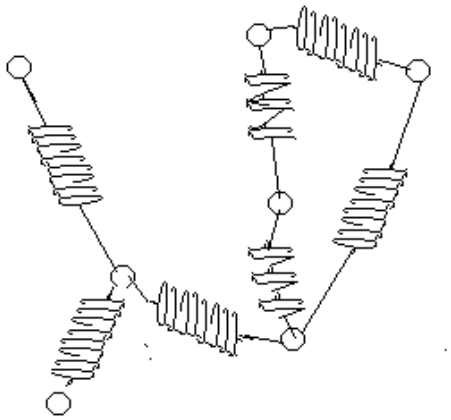

Figure 1: Spring-Mass System

The relationship between the force exerted by the spring and the distance that the spring is stretched is described by Hooke's Law [6] in Equation 1.

$$F = -kx \qquad \text{(Eq. 1)}$$

where F is the restoring force of the object, k is the spring constant, and x is the distance the spring has been extended / compressed from its equilibrium position.

Combined with the damping force (friction) and the acceleration of the object, the new position of an object can be calculated [7].

$$\text{Force} = F_{spring} + F_{damping} = - kx – bv \qquad \text{(Eq. 2)}$$

where b is damping constant and v is velocity.

$$x'' = -kx – bx' \qquad \text{(Eq. 3)}$$

where x is acceleration, x is position of object, and x' is the velocity of the object.

4. Implementation

4.1. Overview

The force-directed placement algorithm described in Section 3 was implemented for the CPU and GPU. The same framework was used for both implementations to process input graph files specified in Graphviz DOT format [8], and store the node and edge information in a two-dimensional array. The placement algorithms for CPU and GPU implementations are run, and the framework writes out the graph information (including placement information) to a DOT file format.

The outer loop of our implemented algorithm is shown in Figure 2.

```
Read_input_graph()
Do {
      Calculate_forces()
      update_velocity()
      update_position()
      Calculate_kinetic_energy()
} While (kinetic energy > stable
threshold);
Write_output_graph()
```

Figure 2: Force-Directed Placement Pseudocode

Calculate Forces: To determine the overall force vector for each node, the force vectors (repulsive and attractive) of all of the other nodes acting upon that node are added together. The complexity of this function is $O(n^2)$

Update Velocities: For each node, the velocity in the x and y direction is calculated based on the force applied to the node for the current timestep, and a damping factor is applied. The complexity of this function is O(n).

Update Position: For each node, the position of the node is updated by adjusting the initial position of the node based on the velocity and timestep. The complexity of this function is O(n).

Calculate Kinetic Energy: The kinetic energy of the system is calculated by summing the square of the velocities in the x and y directions. The complexity of this function is O(n).

In our implementation, we do not strictly emulate a physical system, but rather use a modified set of equations formulated by Eades [2] that considers repulsive forces between all nodes, but only attractive forces between connected nodes.

4.2. CPU Implementation

The CPU implementation of the algorithm is done as a serial application (no multi-threading). The algorithm is implemented as described in Section 4.1.

4.3. GPU Implementation (Basic)

As previously described, the framework processes the input graph and stores the data in a 2-dimensional array. To prepare the data for the GPU implementation, it is transferred into a 1-dimensional array for easier data access during computation.

Our initial implementation began with 2 kernel functions: One kernel to compute forces & update positions and one kernel to calculate the kinetic energy of the system. In this implementation, each thread is responsible for calculating the forces, velocities, and updating the position for one node. Thus, the parallelism of this implementation is limited by the number of nodes in the graph, and the complexity of the calculation. The work per thread is shown in Figure 3.

	Calculate Forces					Calculate Velocity X and Velocity Y	Calculate Position X and Position Y
	Node0	Node1	Node2	Node ...	NodeN		
Node0		T1	T1	T1	T1	T1	T1
Node1	T2		T2	T2	T2	T2	T2
Node2	T3	T3		T3	T3	T3	T3
Node...	T...	T...	T...		T...	T...	T...
NodeN	TN	TN	TN	TN		TN	TN

Figure 3: Basic Implementation (one thread per node)

4.4. GPU Implementation (Increased Parallelism)

The first set of optimizations performed for the GPU implementation were done to re-distribute the work, increase parallelism, and reduce the synchronization overhead of multiple kernel invocations.

4.4.1 Optimization #1: Increasing Parallelism

As described in section 4.1, the Calculate Forces function has a complexity of $O(n^2)$ since an evaluation of all forces between each pair of nodes must be done. In the basic implementation, the parallelism available in the GPU system is not fully exploited. Although the computation for each node is done in parallel, the computation of all of the forces acting on each node is done serially. We increased the parallelism of the GPU implementation to NxN threads (where N is the number of nodes in the system) by having each thread compute the force between one pair of nodes. The work done by each thread is shown in Figure 4.

	Calculate Forces	Calculate Velocity X and Velocity Y	Calculate Position X and Position Y
Node0,1	T1	T1	T1
Node0,2	T2		
Node0,3	T3		
Node0,...	T4		
Node0,N	T5		
Node1,0	T6	T2	T2
Node1,2	T7		
Node1,3	T8		
Node1,...	T9		
Node1,N	T10		
Node...,0	T11	T3	T3
Node...,1	T12		
Node...,2	T13		
Node...,N	T14		
....	T...	T...	T...
NodeN,0	TN-4	TN	TN
NodeN,1	TN-3		
NodeN,2	TN-2		
NodeN,...	TN-1		
NodeN,N-1	TN		

Figure 4: One thread per pairwise node calculation

4.4.2 Optimization #2: Reducing Functional Units

In the basic version of the GPU implementation, the REPULSIVE and ATTRACTIVE force constants are multiplied when determining the force of one node acting upon another. We instead keep these two forces separate and only multiply these constants after all of the force vectors have been summed for each of the repulsive and attractive forces respectively. This reduces the number of functional units that are required during the force computation which is the most expensive: $O(n^2)$.

4.4.3 Optimization #3: Reducing Sync Overhead

In the basic version of the GPU implementation, the calculations for the forces, velocities, and position updates are done in one kernel, and the other kernel computes the kinetic energy. However, before the 2nd kernel can start running, a thread synchronize operation is required to ensure all nodes in the system have been updated before the total kinetic energy of the system can be calculated. To provide more work to each thread, the kinetic energy calculation (sum of the x and y velocities squared) for each node can be done in each thread in the first kernel (avoid thread synchronization). Then the data can be transferred back to the host (synchronization only done once), and a quick summation can be performed for the N data values (one per node in the system). This optimization provided increased parallelism (removed one synchronization barrier), and also improved memory efficiency (reduces number of times the velocity data that is in global memory needs to be read from/written to).

4.5. GPU Implementation (Reduced Memory Bandwidth)

The second set of optimizations performed for the GPU implementation focused on reducing the memory bandwidth requirements by improving memory coalescing, using shared memory, and reducing bank conflicts.

4.5.1 Optimization #4: Improved Memory Coalescing

There were two optimizations performed to memory accesses in the GPU kernel. The first optimization involved combining the float data arrays storing the x and y data for node positions and calculated velocities of the nodes. The x and y data arrays were combined to use the "float2" data type. Although this may incur memory access latency due to potential bank conflicts (two threads in a half warp would be accessing the same memory bank), this allows for improved memory read and write access by issuing one 64-bit data request, rather than two separate 32-bit requests for the x and the y values. In our benchmark measurements, we found that optimization provided a net overall speed-up for the GPU (results shown in Section 6).

The second memory access optimization adjusted the data type for the 2-dimensional array that stored the graph edge weights. The data values stored in this array were 1 if there was an edge between the two nodes and 0 if there was no edge. Hence, the data was stored as type "char". To potentially reduce bank conflicts and improve memory access alignment, we tested the use of an "int" for this data. However, this only showed a negligible speed-up (results shown in Section 6)

4.5.2 Optimization #5: Local Memory

To reduce the number of requests to global memory, we adjusted the GPU implementation to copy data values (that would be accessed multiple times) into local memory. For the GPU kernel function that performs the force calculations, the node position and edge weight data arrays were copied to local memory. The generated forces were stored in local memory until computation was complete and then the data written back to global memory. Similarly, the GPU kernel function that calculates velocity and the next position of each node was updated to use local memory for the velocity, position, and force data arrays.

4.5.3 Optimization #6: Reducing Bank Conflicts

For the GPU implementation that already had increased parallelization optimizations, the computed forces acting on each pair of nodes were stored linearly in a row major array. However, this data layout can incur bank conflicts when the data is accessed in the subsequent GPU kernel which performs the velocity and position calculations. This is because each thread in this second kernel computes the cumulative force of other nodes in the system on a particular node. Hence, it is more efficient to store the data in column major format (so that each thread can be serviced by a different memory bank independently) rather than having a 16-way bank conflict for each data access.

4.5.4 Optimization #7: Reducing Memory Accesses

Optimization #1 increased parallelism by creating one thread per pair of nodes in order to compute the forces. This optimization improved performance but also resulted in increased memory accesses (it was not possible to cache a "source' node position per thread). In an attempt to reduce the memory impact the arrangement of the threads was changed so that each block processed a 16x16 grid of the NxN space. The blocks would first read in the 32 values that were needed first, and then perform the computation without reading any more position data.

4.6. Further GPU Optimizations

Once optimization #7 was done, the time spent in the two kernels per iteration were measured and compared. It was determined that computing the new velocities and kinetic energy now accounted for 80% of the total run time. Additional effort was there for placed in reducing this runtime.

4.6.1 Optimization #8: Using parallel reductions

The majority of the time spent when updating the velocity and position data was summing up all the individual forces computed in the first kernel. To address this problem, a reduction was used where a block was

allocated for each node. Each block consisted of a warp of threads which iterated over the data summing it up before performing the efficient reduction presented in the CUDA example projects.

4.6.2 Optimization #9: Using float4

In an effort to improve memory access efficiency, all the forces were grouped into a single float4 data structure. This data structure could be read and written using a single memory instruction to try to reduce read and write operations.

5. Methodology

The implemented GPU solution was measured in comparison to a CPU implementation of the same force-directed placement algorithm.

5.1 Metrics

There are two metrics that we use to evaluate the quality of our graph placement algorithm: 1) Run Time and 2) Quality of Result.

Run Time:

The runtime speed-up of the GPU implementation is calculated as CPU time / GPU time to achieve the final fully placed graph result.

Quality of Result:

The quality of result is measured by evaluating the final kinetic energy of the system (minimal energy state). The algorithm is deterministic if the same floating point precision is assumed and the arithmetic operations are performed in the same order. However, to fully take advantage of the parallelism available in the GPU, we can relax the requirement for *identical* results to *comparable* quality of results as appreciable to a human user viewing the graph output. The quality of results comparison can be performed by comparing the final result (energy state) of the CPU and GPU. For correctness testing before arithmetic operation re-ordering for performance optimization, the output of the GPU and the CPU can be verified to be identical. For subsequent tests on smaller graphs, the graph output can also be visually inspected to ensure the aesthetic quality of the output is comparable.

5.2 Experimental Procedure

Several large graphs from publicly available sources were used to compare our CPU and GPU implementations. Graphs describing the relationships between actors were generated [10] and graphs describing ISP topology [11] were used to evaluate our implementation. Larger graphs were also synthetically constructed, to help demonstrate the scalability of our GPU implementation, and show the effect of each optimization. Some of the larger graphs were too complex for the CPU implementation to finish running (excessively long run times). For the large graphs, we also verified that the placement algorithms included with the Graphviz DOT package also had difficulties processing these graphs.

All evaluations were performed using the CUDALAB machines running Debian 4.0. Each machine has an Intel Core 2 Quad Q9550 2.83GHz quad-core CPU, 4GB of RAM, and an NVIDIA GTX280 GPU with 1GB of memory. The runtimes were recorded for the CPU and all GPU implementations.

6. Evaluation

The CPU implementation and the results from the GPU implementation (with optimizations) were evaluated using the procedure described in Section 5. The speed-up obtained by the GPU implementation vs. the CPU implementation is shown in Figure 5.

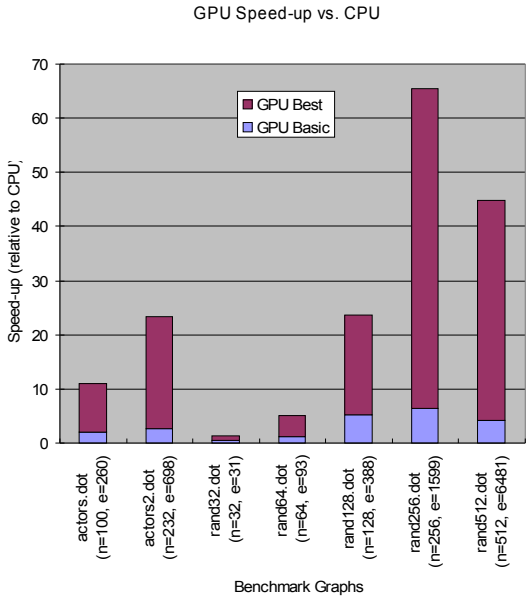

Figure 5: GPU Speed-up vs. CPU

By porting the Force-Directed Placement algorithm directly to the GPU (basic implementation), there is a slowdown of 55% for the random graph rand32.dot (32 nodes, 31 edges) and an up to 6x speed-up achieved for the random graph rand256 (256 nodes, 1599 edges). For two "real" graphs describing relationships between actors, a 2x speed-up was achieved for actors.dot (100 nodes, 260 edges) and a 2.7x speed-up for actors2.dot (232 nodes, 698 edges).

After optimizing the GPU implementation, there remained no speed-up for the random graph rand32.dot, and a 58x speed-up was achieved for rand256.dot. The two "real" actor graphs achieved 9x for actors.dot and 20.6x speed-up for actors2.dot.

The GPU implementation is unable to provide a net speed-up for the rand32.dot graph because there are so few nodes in the graph (limited parallelism to exploit), and very limited connectivity between the nodes (requires many iterations to achieve system equilibrium since there are very few attractive forces).

To evaluate the scalability of the GPU implementation, we increased our benchmark design set by synthetically generating graphs. The time to process the benchmark graphs vs. graph size (number of nodes + number edges) are shown in Figure 6. The CPU run times increase quadratically with design size, since the calculation of the forces is of complexity $O(n^2)$. For small graph sizes, the GPU implementation provides moderate to no speed-up, but provides a significant speed-up for large graph sizes since there is more work to be parallelized. The GPU implementation exhibits a linear run time increase (with very moderate slope) since it is able to increase parallelism by taking advantage of the processors available with only a small overhead.

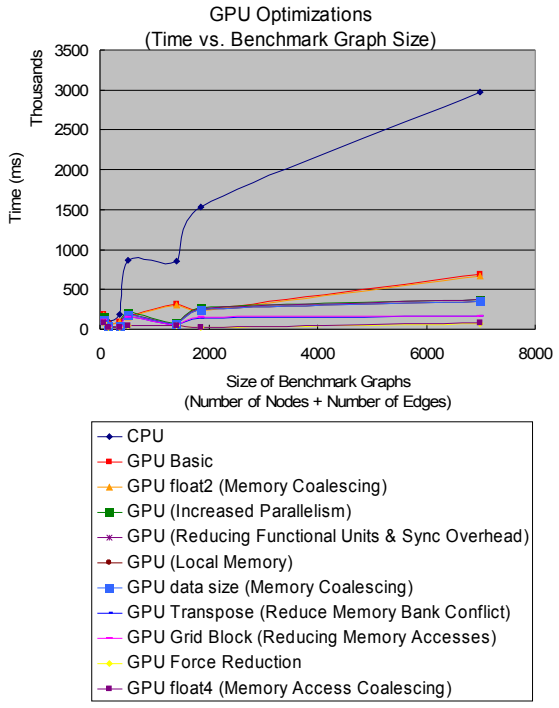

Figure 6: GPU Optimizations (Time vs. Graph Size)

The percentage run time improvement for each optimization (as compared to the previous optimization) is shown in Figure 7. The optimizations that provided the most significant speed-ups (in chronological order based on application to the GPU implementation) were:

1) Porting directly to the GPU
2) GPU Increased Parallelism
3) GPU Transpose (Reducing Memory Bank Conflicts)
4) GPU Force Reduction

Several of the memory coalescing optimizations resulted in no speed-up (or even a slight increase in run time):

1) GPU Data Size (Memory Coalescing)
2) GPU Grid Block (Reducing Memory Accesses)
3) GPU Float4 (Memory Access Coalescing)

	rand32 (n=32, e=31)	actors (n=100, e=260)	actors2 (n=708, e=698)	rand256 (n=256, e=1599)
CPU	0%	0%	0%	0%
GPU Basic	**-55%**	97%	171%	538%
GPU float2 (Memory Coalescing)	5%	5%	5%	5%
GPU (Increased Parallelism)	22%	79%	281%	**-15%**
GPU (Reducing Functional Units & Sync Overhead)	23%	11%	5%	4%
GPU (Local Memory)	4%	4%	6%	4%
GPU Data Size (Memory Coalescing)	0%	0%	**0%**	**0%**
GPU Transpose (Reduce Memory Bank Conflict)	6%	13%	18%	73%
GPU Grid Block (Reducing Memory Accesses)	**-4%**	1%	**-1%**	**-2%**
GPU Force Reduction	31%	83%	46%	469%
GPU Float 4 (Memory Access Coalescing)	2%	**-1%**	**-10%**	**-3%**

Figure 7: GPU Optimizations (% Runtime Improvement)

It is speculated that these optimizations did not provide a net benefit to improve run time because there was already sufficient work for each thread that the memory latency could be hidden. Increased memory coalescing actually could lead to a worsening of performance in some cases, since the optimizations increased the possibility of memory bank access conflicts.

The speed-up contribution from each GPU algorithm optimization is shown in Figure 8. The initial port of the CPU algorithm to the GPU achieved a moderate speed-up, and subsequent optimizations to increase parallelism and improve memory accesses provided significant performance improvements.

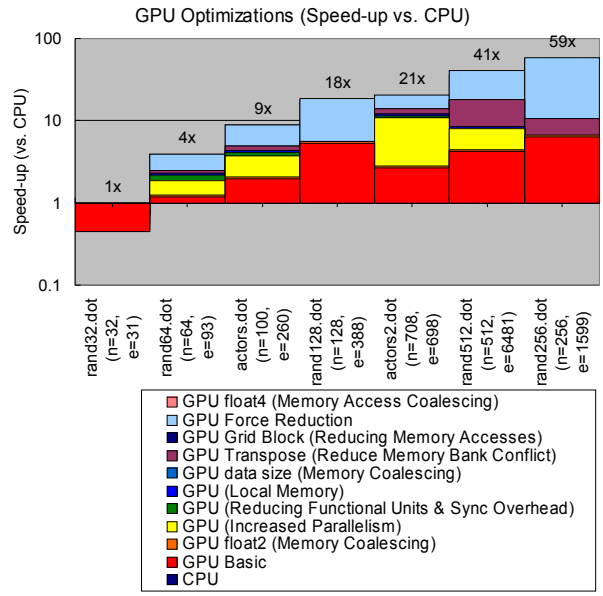

Figure 8: GPU Optimizations (Speed-up vs. CPU)

The quality of the resulting output graphs were visually compared for the CPU and the GPU implementations. In all cases the output graphs generated by the GPU are very similar to those generated by the CPU. The positions of some nodes differ by a few pixels, which is believed to be rounding errors of different ALUs in the CPU and GPU. Nevertheless, the placement of each node remains the same relative to all other nodes, and it does not affect the aesthetic quality. Figure 9 shows the layout of the ISP topology graph (300 nodes, 549 edges) generated by the GPU implementation.

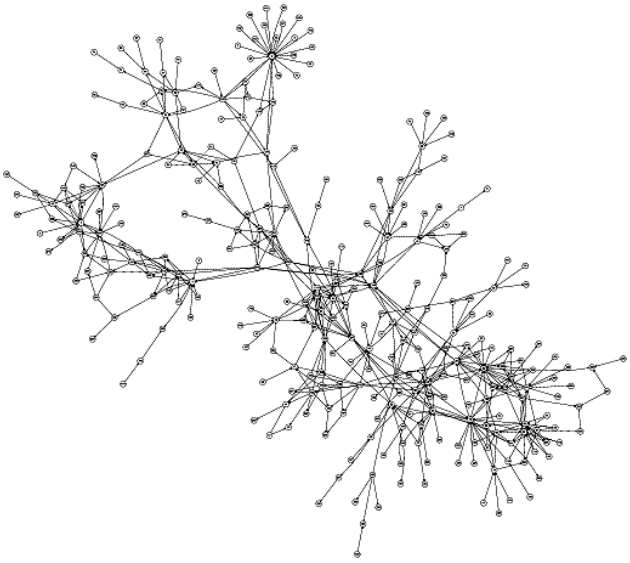

Figure 9: Layout of ISP topology graph "top1755.dot"

7. Conclusions

The results of our investigation have shown that a GPU implementation of a graph placement & layout algorithm can significantly improve the runtime and achieve comparable quality to a CPU implementation. The GPU implementation was able to achieve up to a 58x speed-up with no degradation in the quality of graph placement output.

Future potential extensions for this work could include implementation of a simulated annealing algorithm (as described in Section 3. A simulated annealing algorithm lends itself well to a parallelization (many prospective moves can be evaluated in parallel). However, it is unclear how long it may take to converge on a "good" solution, and what temperature annealing schedule would be appropriate to ensure local minima were avoided.

The graph placement algorithms included with the Graphviz tool suite have difficulty supporting moderately sized graphs and have a long runtime. The current GPU implementation of the force-directed placement algorithm could be ported to interface with the Graphviz tool suite (currently supports different CPU placement algorithms), so that when a CUDA compatible device is available, it would provide a speed-up for the DOT graph processing and drawing application.

8. References

[1] "Graph drawing" [Online] Available: http://en.wikipedia.org/wiki/Graph_drawing

[Accessed: April 5, 2009].

[2] P. Eades, 'A heuristic for graph drawing', Congressus Nutnerantiunt, 42, 149–160 (1984).

[3] T. M. J. Fruchterman, E. M. Reingold: Graph drawing by force-directed placement. Software.

Practice & Experience 21, 11 (1991), 1129.1164. 3, 4

[4] Y. Frishman, A. Tal 'Online Dynamic Graph Drawing' IEEE-VGTC Symposium on Visualization (2007)

[5] Y. Frishman, A. Tal 'Multi-Level Graph Layout on the GPU'. Available: http://www.ee.technion.ac.il/~ayellet/Ps/FrishmanTalInfoVis07.pdf [Accessed: April 2009]

[6] "Hooke's Law" [Online] Available: http://en.wikipedia.org/wiki/Hooke's_law [Accessed April 9, 2008]

[7] 'MyPhysicsLab – Single Spring' [Online] http://www.myphysicslab.com/spring1.html [Accessed April 9th 2009]

[8] GraphViz [Online] Available : http://www.graphviz.org/ [Accessed March 14th 2009]

[9] Proximity Graphs [Online] Available: http://www.research.att.com/~volinsky/cgi-bin/prox/prox.pl [Accessed: April 5th 2009]

[10] Rocketfuel: An ISP Topology Mapping Engine [Online] Available: http://www.cs.washington.edu/research/networking/rocketfuel/ [Accessed April 9th 2009]