Speeding up Network Layout and Centrality Measures with NodeXL and the Nvidia CUDA Technology

(10/11/2010)

Puneet Sharma[1, 2], Udayan Khurana[1, 2], Ben Shneiderman[1, 2], Max Scharrenbroich[3], and John Locke[1]

[1]Computer Science Department &
[2]Human-Computer Interaction Lab &
[3]Department of Mathematics
University of Maryland
College Park MD 20740
{puneet,udayan,ben}@cs.umd.edu, (john.b.locke, max.franz.s)@gmail.com

Abstract

In this paper we talk about speeding up calculation of graph metrics and layout with *NodeXL* by exploiting the parallel architecture of modern day Graphics Processing Units (GPU), specifically *Compute Unified Device Architecture (CUDA)* by *Nvidia*. Graph centrality metrics like *Eigenvector, Betweenness, Page Rank* and layout algorithms like *Fruchterman-Rheingold* are essential components of *Social Network Analysis (SNA)*. With the growth in adoption of SNA in different domains and increasing availability of huge networked datasets for analysis, social network analysts are looking for tools that are faster and more scalable. Our results show up to 802 times speedup for a Fruchterman-Rheingold graph layout and up to 17,972 times speedup for Eigenvector centrality metric calculations.

1. Introduction

NodeXL[1] [3] is a network data analysis and visualization [10] plug-in for Microsoft Excel 2007 that provides a powerful and simple means to graph data contained in a spreadsheet. Data may be imported from an external tool and formatted to NodeXL specifications, or imported directly from the tool using one of the supported mechanisms (such as importing a Twitter network). The program will map out vertices and edges using a variety of layout algorithms, and calculate important metrics on the data to identify nodes of interest.

While NodeXL is capable of visualizing a vast amount of data, it may not be feasible to run its complex computation algorithms on larger datasets using hardware that is typical for a casual desktop user. For example, while visualizing the voting patterns of 100 senators in Congress is achievable even to low-powered PCs, larger datasets (like those found in Stanford's SNAP library[2]) may contain millions of nodes, requiring more substantial hardware. We believe that in order for data visualization tools to achieve popularity, it is important that they are accessible and fast for users using average desktop hardware.

[1] NodeXL: Network Overview, Discovery and Exploration for Excel http://nodexl.codeplex.com/
[2] SNAP: Stanford Network Analysis Platform http://snap.stanford.edu/index.html

Nvidia's CUDA[3] technology provides a means to employ the dozens of processing cores present on modern video cards to perform computations typically meant for a CPU. We believe that CUDA is appropriate to improve the algorithms in NodeXL due to the abundance of Graphical Processing Units (GPUs) on mid-range desktops, as well as the parallelizable nature of data visualization. Using this technology, we hope to allow users to visualize and analyze previously computationally infeasible datasets.

In our modified NodeXL implementation, we targeted two computationally expensive data visualization procedures: the Fruchterman-Rheingold [1] force-directed layout algorithm, as well as the eigenvector centrality node metric.

2. Computational Complexity of SNA Algorithms

Social Network Analysis consists of three major areas - Network Visualization, Centrality Metric Calculation and Cluster detection. Popular techniques for Visualization use Force-Directed algorithms to calculate a suitable layout for nodes and edges. Computationally, these algorithms are similar to n-body simulations done in physics. The two most popular layout algorithms are Harel-Koren [2] and Fruchterman-Rheingold [1]. The computational complexity of the latter can roughly be expressed as O(k (V2+E)) and the memory complexity as O(E+V), where k is the number of iterations needed before convergence, V is the number of vertices and E is the number of edges in the graph. Centrality algorithms, calculate the "relative importance of each node with respect to rest of the graph". For example, betweenness centrality [8] tells how "in-between" a node is within a complete graph by measuring the number of all shortest paths that pass through that node. The best known algorithm for that takes O(VE + V2logV) time and O(N+V) memory [5]. Eigenvector centrality [8] measures the importance of a node by the measure of its connectivity to other "important" nodes in the graph. The process of finding it is similar to belief propagation and the algorithm is iterative with the time complexity O(k.E).

In dense graphs, the number of edges E, tends to approach O(V2). With the increase in the E and the V value, the computation becomes super-linearly expensive. For example, to calculate Betweenness centrality and eigenvector centrality for a graph with V=2,625, and E=99,999 took 570 s and 23 s respectively. And for a graph with V=17,425, E=1,785,056 it took 40,265 s and 9,828 s respectively. These results were obtained with NodeXL on an Intel Core Duo 3.0 GHz, 3.25 GB RAM, WinXP machine. In either commercial or academic worlds, an analysis of a graph with 18k nodes is a reasonable task, where as a response time of 12 minutes for Betweenness centrality calculation alone isn't. For that matter, networks with over 100,000 nodes are a very common these days. Therefore, speedup and scalability are key challenges to SNA.

Speedup and scalability are two issues that seem very closely connected, but one does not necessarily imply the other. A program scales to an input size if it can successfully execute inputs of that size within a reasonable time. Speedup is essentially concerned with decreasing the time taken per unit of the input. A highly scalable program might take more time than another program that does not scale to larger inputs but runs faster on inputs below a certain size.

2.1 Approaches for Speedup and Scalability

For speedup, two approaches that immediately come to mind are faster algorithms with better implementation and secondly, more hardware to compute in parallel. If we believed that the best algorithms (possible, or the best known) are already in place, the second area is worth investigating.

[3] What is CUDA? http://www.nvidia.com/object/what_is_cuda_new.html

Extra hardware to speedup the process could be done in a distributed or parallel environment. Distributed computation works on the principle of cheaper loosely coupled hardware [7], where as parallel computation works on tightly coupled shared-memory architecture [Supercomputers], but is more expensive for the same amount of computational power. Since they are loosely coupled, the distributed systems have a bigger overhead of inter-node communication. Since distributed architectures are much cheaper and the architecture is much more scalable, from a purely scalability perspective, distributed computing is ideal in theory. However, it is the nature of the algorithm which determines that whether it can be executed effectively in a distributed environment or not. If the graph can be easily partitioned into separate workable sets, then the algorithm is suitable for distributed computing, otherwise not. This is because the communication between nodes in such an environment is minimal, and so all the required data must be present locally. We divide the set of algorithms used in SNA into three different categories based on feasibility of graph partitioning:

- Easy Partitioning - Algorithm requires only the information about locally connected nodes (neighbors) e.g. Vertex Degree Centrality, Eigenvector centrality
- Relatively Hard Partitioning - Algorithm requires global graph knowledge, but local knowledge is dominant and efficient approximations can be made. e.g. Fruchterman-Rheingold
- Hard Partitioning - Data partitioning is not an option. Global graph knowledge is required at each node. e.g. Betweenness centrality which requires calculation of All Pairs of Shortest Paths.

Algorithms that fall under the first category can be made to scale indefinitely using a distributed system like Hadoop [9]. For the second category, there is a possibility of a working approximation algorithm, whereas nothing can be said about the third category of algorithms.
Parallel systems on the other hand consist of a shared memory, high speed inter-processor communication architecture. So, such a system is ideal for large speedups. While the parallelization on such a system may seem easier and more efficient, such hardware is limited in capacity due to cost. A supercomputer is more costly than a Hadoop cluster with same cumulative processing power. Hence scalability of even the first class of algorithms is not trivial. In the next section we discuss an implementation of Eigenvalue Centrality that gives considerable speedups and arguably scalability as well.

3. Layout Speedup

3.1. Layout Enhancement

NodeXL includes about a dozen different layout algorithms to display a user's data. Many of them are based on simple geometric formations (Spiral, Sine Wave, etc), and do not provide an interesting view of data in most cases. Two algorithms, *Fruchterman-Rheingold* and *Harel-Koren Fast Multiscale* provide more interesting views for various data types. The Fruchterman-Rheingold algorithm was chosen as a candidate for enhancement due to its popularity as an undirected graph layout algorithm in a wide range of visualization scenarios, as well as the parallelizable nature of its algorithm. The algorithm works by placing vertices randomly, then independently calculating the attractive and repulsive forces between nodes based on the connections between them specified in the Excel workbook. The total kinetic energy of the network is also summed up during the calculations to determine whether the algorithm has reached a threshold at which additional iterations of the algorithm are not required.

The part of the algorithm that computes the repulsive forces is $O(N^2)$, indicating that it could greatly benefit from a performance enhancement (the rest of the algorithm is $O(|V|)$ or $O(|E|)$, where v and e

are the vertices and edges of the graph, respectively). The algorithm is shown below in pseudo-code and computationally intensive repulsive calculation portion of the algorithm is italicized [1]:

```
area := W * L; { W and L are the width and length of the frame }
G := (V, E); { the vertices are assigned random initial positions }
k := sqrt(area/|V|);
function fa(z) := begin return $x^2/k$ end ;
function fr(z) := begin return $k^2/z$ end ;

for i := 1 to iterations do begin
        { calculate repulsive forces}
        for v in V do begin
                { each vertex has two vectors: .pos and .disp }
                v.disp := 0;
                for u in V d o
                        if (u != v) then begin
                                { D is short hand for the difference}
                                { vector between the positions of the two vertices )
                                D  := v.pos - u.pos;
                                v.disp := v.disp + ( D /| D |) * fr (| D |)
                        end
                end
        end
        { calculate attractive forces }
        for e in E do begin
                { each edge is an ordered pair of vertices .v and .u }
                D  := e.v.pos – e.u.pos
                e.v.disp := e.v.disp – ( D/|  D |) * fa (| D |);
                e.u. disp := e.u.disp + ( D /| D |) * fa (| D |)
        end
        { limit the maximum displacement to the temperature t }
        { and then prevent from being displaced outside frame}
        for v in V do begin
                v.pos := v.pos + ( v. disp/ |v.disp|) * min ( v.disp, t );
                v.pos.x := min(W/2, max(-W/2, v.pos.x));
                v.pos.y := min(L/2, max(–L/2, v.pos.y))
        end
        { reduce the temperature as the layout approaches a better configuration }
        t := cool(t)
end
```

Due to the multiple nested loops contained in this algorithm, and the fact that the calculations of the forces on the nodes are not dependent on other nodes, we implemented the calculation of these forces in parallel using CUDA. In particular, the vector *V* was distributed across GPU processing cores. The resulting algorithm, **Super Fruchterman-Rheingold**, was then fit into the NodeXL framework as a selectable layout to allow users to recruit their GPU in the force calculation process as shown below:

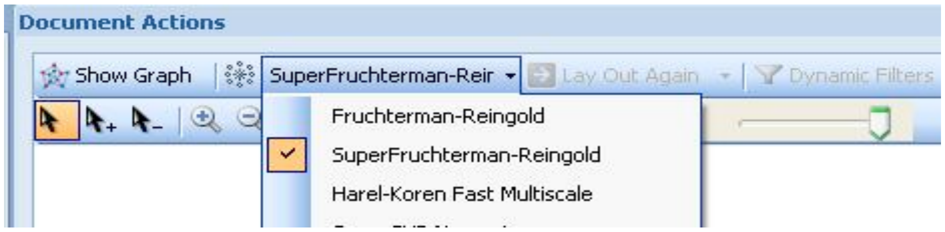


Figure 1: NodeXL with Super Fruchterman-Rheingold Layout

3.2. Results for layout speedup

We ran our modified NodeXL on the following hardware configuration:

Computer Hardware	**Description**
GPU	GeForce GTX 285, 1476 MHz, 240 cores
Host CPU	3 GHz, Intel(R) Core(TM)2 Duo

Table 1: Hardware configuration for layout speedup testing

This machine is located in the Human Computer Interaction Lab at the University of Maryland, and is commonly used by researchers to visualize massive network graphs. The results of our layout algorithm are shown in the table below. Each of the graph instances listed may be found in the Stanford SNAP library:

Graph Instance Name	**#Nodes**	**#Edges**	**Super F-R run time (ms)**	**Simple F-R run time (ms)**	**Speedup**
CA-AstroPh	18,772	396,160	1,062.4	84,434.2	**79x**
cit-HepPh	34,546	421,578	1,078.0	343,643.0	**318x**
soc-Epinions1	75,879	508,837	1,890.5	1,520,924.6	**804x**
soc-Slashdot0811	77,360	905,468	2,515.5	1,578,283.1	**625x**
soc-Slashdot0902	82,168	948,464	2,671.7	1,780,947.2	**666x**

Table 2: Results of layout speedup testing

The results of parallelizing this algorithm were actually better than we had expected. The speedup was highly variable depending on the graph tested, but the algorithm appears to perform better with larger datasets, most likely due to the overhead of spawning additional threads. Interestingly, the algorithm also tends to perform better when there are many vertices to process, but a fewer number of edges. This behavior is shown below:

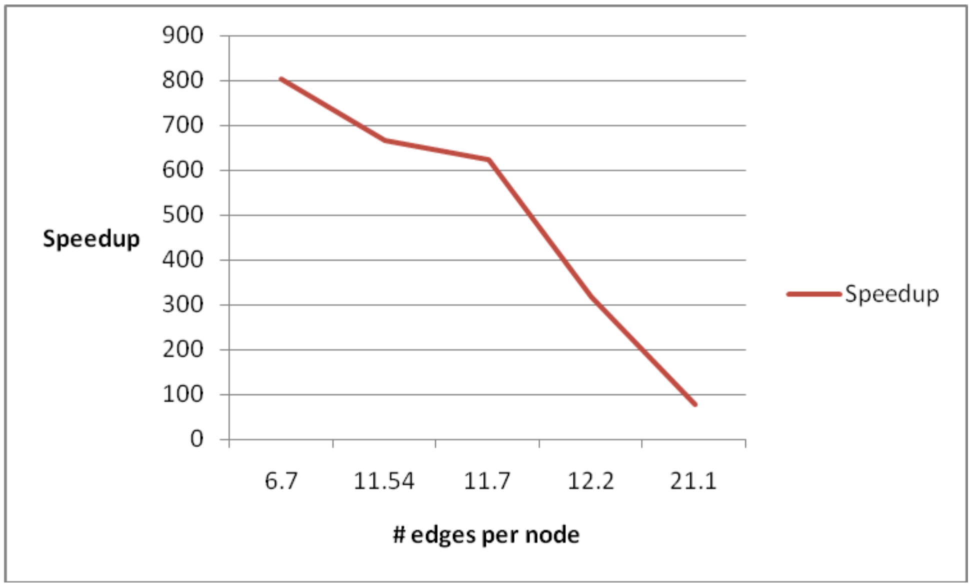


Figure 2: Graph showing Layout Speedup and number of edges per node

The *soc-Epinions1* dataset contains relatively few edges for the number of vertices in the graph (1:6.7 vertex to edge ratio), leading to a 804x speedup. However, while the *cit-HepPh* dataset contains a similar number of edges, it contains only half the vertices (1:12.2 ratio). As a result, there is only a 318x speedup.

This behavior is the result of reaping the benefits of many processing cores with a large number of vertices, while the calculation of the repulsive forces *at each node* still occurs in a sequential fashion. A graph with myriad nodes with no edges would take a trivial amount of time to execute on each processor, while a graph with few vertices and myriad connections between them would still require much sequential processing time. Regardless, the Super Fruchterman-Rheingold algorithm performs admirably even in the worst case of our results, providing a 79x speedup in the *CA-AstroPh* set with a 1:21 vertex to edge ratio. The ability for a user to visualize their data within two seconds when it originally took 25 minutes is truly putting a wider set of data visualization into the hands of typical end users.

4. Metric Speedup

4.1. Metric Enhancement

The purpose of data visualization is to discover attributes and nodes in a given dataset that are interesting or anomalous. Such metrics to determine interesting data include *betweenness centrality, closeness centrality, degree centrality,* and *eigenvector centrality*. Each metric has its merits in identifying interesting nodes, but we chose to enhance eigenvector centrality, as it is frequently used to determine the “important” nodes in a network. Eigenvector centrality rates vertices based on their connections to other vertices which are deemed important (connecting many nodes or those that are “gatekeepers” to many nodes).

The following formula defines the calculation algorithm for eigenvector centrality:

$$x_i = \frac{1}{\lambda_{max}} \sum_{j=1}^{N} x_j A_{ij}$$

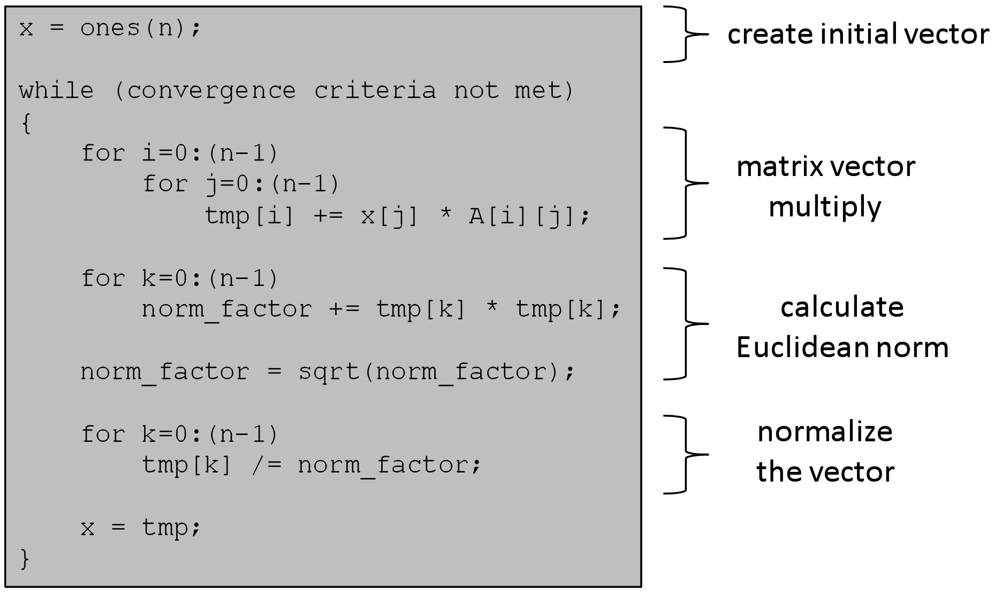
For $i = 1, \dots, N$, where x_i is the eigenvector centrality of vertex i, A_{ij} is the adjacency matrix of the graph and λ_{max} is the largest eigenvalue of the adjacency matrix. In matrix form we have:

$$\boldsymbol{x}\lambda_{max} = \boldsymbol{Ax}$$

By the Perron-Frobenius theorem we know that for a real valued square matrix (like an adjacency matrix) there exists a largest unique eigenvalue that is paired with an eigenvector having all positive entries.

4.2. Power Method

An efficient method for finding the largest eigenvalue/eigenvector pair is the power iteration or power method. The power method is an iterative calculation that involves performing a matrix-vector multiply over and over until the change in the iterate (vector) falls below a user supplied threshold. We outline the power method algorithm below with the following pseudo-code:



```
x = ones(n);

while (convergence criteria not met)
{
    for i=0:(n-1)
        for j=0:(n-1)
            tmp[i] += x[j] * A[i][j];

    for k=0:(n-1)
        norm_factor += tmp[k] * tmp[k];

    norm_factor = sqrt(norm_factor);

    for k=0:(n-1)
        tmp[k] /= norm_factor;

    x = tmp;
}
```

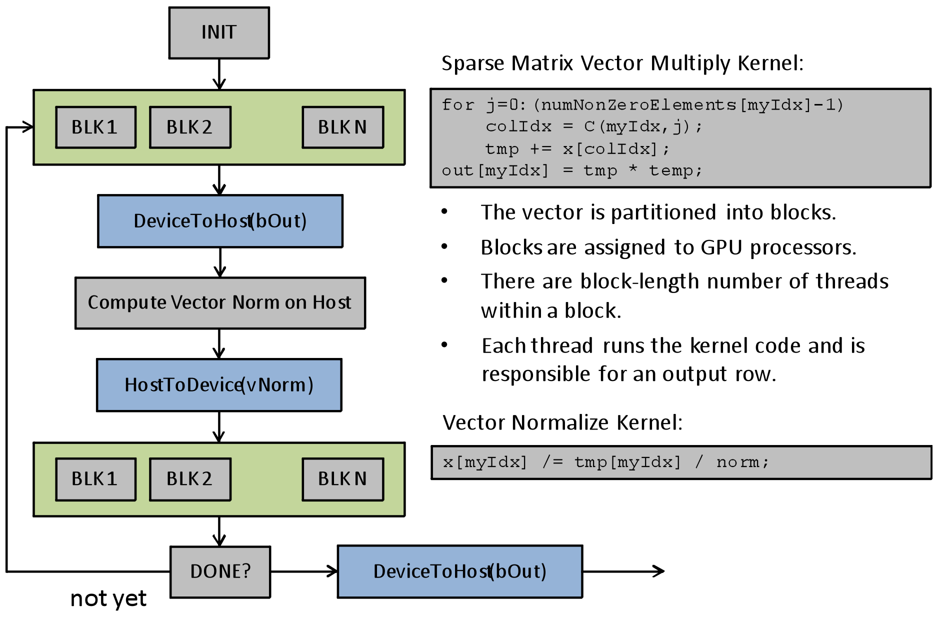
Figure 3: Power Method algorithm

In the initialization step, the starting iterate is set to a vector of all ones, allowing the algorithm to behave deterministically, a useful property for performance analysis. Following initialization, the algorithm enters a while loop consisting of a matrix-vector multiply, followed by a calculation of the iterate's Euclidean norm, and finally a vector normalization. Interesting graphs tend to be sparse which means the adjacency matrix will generally be sparse, and can be efficiently represented in compressed row storage (CRS). In addition to being space efficient, storing the matrix in CRS improves the efficiency of the matrix-vector multiply in the loop. The pseudo-code for the sparse matrix-vector multiply is:

```
// sparse matrix vector multiply
for i=0:(n-1)
     for j=0:(numNonZeroElements[i]-1)
          colIdx = C(I,j);
          tmp[i] += x[colIdx];
```

4.3. Power Method on a GPU

For our implementation we divided the computation between the host and the GPU. A kernel for matrix-vector multiply is launched on the GPU, followed by a device-to-host copy of the resulting vector for computation of the vector norm, then a host-to-device copy of the vector norm and finally a kernel for vector normalization is launched on the GPU. Below is a figure outlining the algorithm flow to allow execution on the GPU:



Sparse Matrix Vector Multiply Kernel:

```
for j=0:(numNonZeroElements[myIdx]-1)
     colIdx = C(myIdx,j);
     tmp += x[colIdx];
out[myIdx] = tmp * temp;
```

- The vector is partitioned into blocks.
- Blocks are assigned to GPU processors.
- There are block-length number of threads within a block.
- Each thread runs the kernel code and is responsible for an output row.

Vector Normalize Kernel:

```
x[myIdx] /= tmp[myIdx] / norm;
```

Figure 4: Algorithm flow

Although we could have implemented the summation portion of the vector norm in the GPU, we found this changed the order of operations, resulting in different convergence behavior of the algorithm. We saw both increases and decreases (sometimes significant) in the number of iterations until convergence for the range of problem instances we studied. We felt that it was important to maintain the behavior of the algorithm in both the host-only and the GPU version for testing and validation purposes. While parallel vector summation was not an option in our case, we were still able to keep most of the vector norm computation on the GPU. We did this by computing part of the vector norm in the matrix-vector multiply kernel at the end of the kernel by squaring the output vector element. Therefore, computing the norm on the host only involved a vector sum followed by a square root, a relatively fast operation on the host CPU[4].

[4] In our testing we found that the time spent computing the rest of the vector norm on the host did not significantly affect the

4.4. Results for Eigenvector Centrality speedup

We tested our GPU algorithm on eleven graph instances of up to 19k nodes and 1.7m edges. As a comparison, we used the serial version of the algorithm implemented in NodeXL with C#. Below is a table showing the hardware on which the algorithms were run:

Computer Hardware	**Description**
GPU	GeForce GTX 285, 1476 MHz, 240 cores
Host CPU	3 GHz, Intel(R) Core(TM)2 Duo

Table 3: Hardware configuration for Eigenvector centrality metric speedup testing

Below is a table and a plot showing the speedups we achieved by running the algorithm on the GPU.

Graph Name	**#Nodes**	**#Edges**	**Time - GPU(sec)**	**Time - CPU (sec)**	**Speedup**
Movie Reviews	2,625	99,999	0.1874928	23.0	**123x**
CalTech Facebook	769	33,312	0.0156244	1.6	**102x**
Oklahoma Facebook	17,425	1,785,056	0.5468540	9,828.0	**17972x**
Georgetown Facebook	9,414	851,278	0.1874928	1,320.0	**7040x**
UNC FB Network	18,163	1,533,602	1.8749280	13,492.0	**7196x**
Princeton FB Network	6,596	586,640	0.093746	495.0	**5280x**
Saket Protein	5,492	40,332	0.109370	107.0	**798x**
SPI - Human.pin	8,776	35,820	0.078122	263.0	**3366x**
Wiki Vote	7,115	103,689	0.265614	137.0	**516x**
Gnutella P2P	10,874	39,994	0.171868	681.7	**3966x**
AstroPh Collab	18,772	396,160	0.3437368	2,218.7	**6454x**

Table 4: Tabular view of the speedups

Speedups ranged between 102x and 17,972x. At first glance there does not seem to be a clear relationship between the problem instances and the speedups. We further investigated this to try and explain the results. We first examined how problem size, measured by the number of edges, affected the speedup as shown below:

overall time of the algorithm.

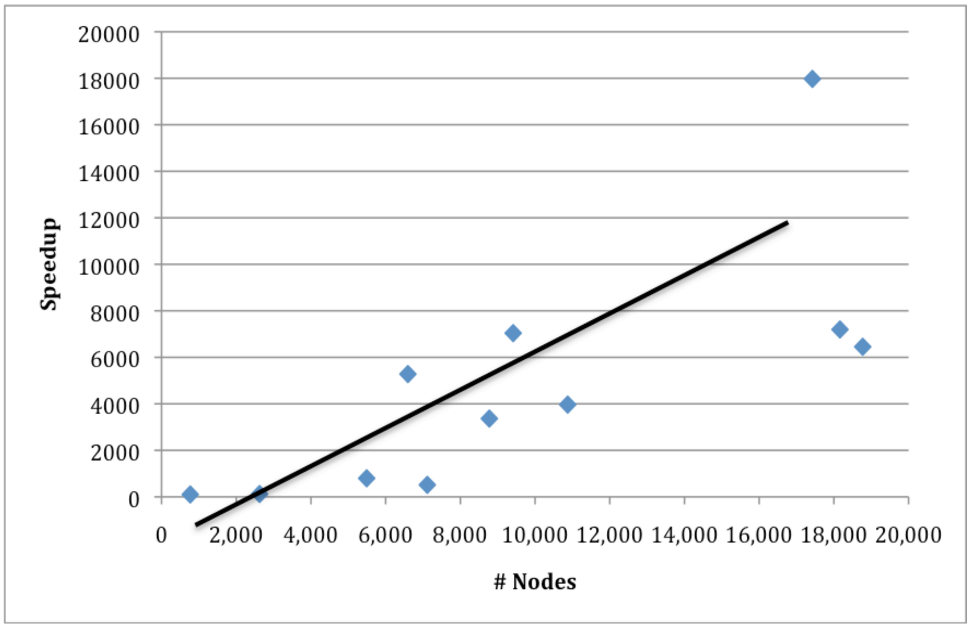


Figure 5: Graph showing Eigenvector Centrality Speedup and number of edges

There is an increase in the speed up as the size of the graph increases. We believe that this is due to the effects of scale on the serial CPU implementation for graphs with bigger sizes.

4.5. Scalability

As the results in the next section will show, CUDA helps us achieve huge speedups for Eignevector Centrality on small graphs. Let us now discuss the scalability of a CUDA program as the small device memory is an obvious bottleneck. An efficient representation of the graph is essential in scaling to as large graphs as possible. However, even that does not guarantee scalability to arbitrarily large graphs. This is because beyond some size of the graph, the device memory is going to fall short. So, we need a scheme where we can partition the data such that we can load, compute on and unload the various partitions one by one, without the loss of information. The case where such a partitioning is possible is an ideal one but it really depends on the algorithm. We show below that it is possible for Eigenvector centrality. We assume that the main memory has enough heap storage of at least twice the size needed to hold the graph.

```
AdjList := Adjacency List for Graph, G
Max_GPU_Size := maximum size for which the GPU memory can hold the graph G.
P := ceiling (Sizeof(G)/Max_GPU_Size) // Number of Partitions
Make P Pseudo pointer representations of partitions of G, such that no partition size > Max_GPU_Size
EVC_values_new[n], EVC_values_old[n]
while ( Convergence is not achieved for EVC values)
    LoadIntoDeviceMemory(EVC_values_new, EVC_values_old)
    for (i := 0 ; i < P ; ++i)
        LoadIntoDeviceMemory(Partion i,)
        Run EVC algo on GPU
        DeleteFromDeviceMemory(Partition i)
    end
    EVC_values_new = EVC_old_values
    DeleteFromDeviceMemory(EVC_values_new, EVC_values_old)
End
```

This arguably provides a scalable Eigenvector Centrality program. As long as the main memory can contain the graph information, chunks of the graph can be swapped to and from with the device memory. However, if the graph is too large to be held at once in the main memory, more needs to be done. Use of virtual memory or an application level *file i/o* can ensure that the memory demand never exceeds its capacity. Although disk storage access is considerably slower than the physical or device memory, the data transfer happens only in large chunks at intervals of time. This hypothesis, however, needs to be tested for performance in practice for very huge graphs, only a minor fraction of which can be loaded into the memory at once. The hierarchy of storage can be viewed as analogous to different levels of caching. The actual scaled out implementation is out of the scope of this paper and we propose it as an area of the future work.

5. Future Work

One facet of this work that we hope to discover more about in the future is the scalability of the CUDA Super Fruchterman-Rheingold layout algorithm. We were fortunate to have large datasets available from the SNAP library, but given the 666x speedup obtained in the largest dataset tested, which contains 82,163 nodes, we have yet to determine the limits of the algorithm. It is clear that given the restricted amount of memory on today's GPUs (about 1 GB on high-end cards) that eventually memory will become a bottleneck, and we have yet to determine where that limit occurs. There are obviously interesting networks that contain millions of nodes, and we hope to be able to provide significant speedup for those situations as well. Unfortunately, the Windows Presentation Foundation code which paints a graph on the visualization window proved to be the weakest link in the visualization of mult-million node graphs. On large enough graphs, the WPF module will generate an out-of-memory error or simply not terminate. Overcoming this WPF shortcoming would be the first step towards finding the bounds of the Super Fruchterman-Rheingold algorithm.

Additionally, there are other algorithms within NodeXL that could benefit from GPU-based parallelization. The Harel-Koren Fast Multiscale algorithm [2] is another heavily-used layout algorithm in data visualization that is currently implemented sequentially. There is a definite advantage to enhancing a multiscale force-directed layout algorithm to run in the time we achieved with the Fruchterman-Rheingold enhancement, although given the flow dependency issues in the algorithm, it could prove to be difficult to parallelize and might not achieve the same speedup.

We also leave the option open to perform additional portions of the algorithms we modified on the GPU, such as the less CPU-intensive tasks in Fruchterman-Rheingold and the summation portion of the vector norm in the eigenvector centrality calculation. While delegating this part of the eigenvector centrality calculation proved to be problematic due to the modification of order of operations, parallelization of this portion remains a possibility.

Finally, the speedup achieved using the eigenvector centrality metric also bring other vertex metrics to our attention. In particular, *closeness* and *betweenness* centrality are two measures that are used frequently in data visualization, and have yet to be implemented using CUDA. Considering each metric is calculated at individual nodes, these enhancements should not be particularly difficult to implement as well.

6. Conclusions

NodeXL is a popular data visualization tool used frequently by analysts on typical desktop hardware. However, datasets can be thousands or millions of nodes in size, and a typical desktop CPU does not contain enough processing cores to parallelize the various algorithms contained in NodeXL sufficiently to execute in a reasonable timeframe. Nvidia's CUDA technology provides an intuitive computing interface for users to use the dozens of processing elements on a typical desktop GPU, and we applied the technology to two algorithms in NodeXL to show the benefit that the application can gain from these enhancements.

The results obtained from running these enhancements were impressive, as we observed a 79x - 804x speedup in our implemented layout algorithm. Further, the 102x - 17,972x speedup gained by parallelizing the eigenvector centrality metric across GPU cores indicates that when a CUDA-capable GPU is available, it should be used to perform centrality metrics. Such improvement in these execution times brings the visualization of data previously infeasible on a standard desktop into the hands of anyone with a CUDA-capable machine, which includes low-to-middle end GPUs today. While we have by no means maxed out the parallelization possibilities in the NodeXL codebase, we hope to have provided enough preliminary results to demonstrate the advantage that CUDA technology can bring to data visualization.

8. Acknowledgements

We would like to acknowledge Dr. Ben Shneiderman for continuously motivating us to consider CUDA as a possible means for addressing the speed and scalability issues in Social Network Analysis. We also thank Dr. Amitabh Varshney and Dr. Alan Sussman for their guidance through the course of this project. We are grateful to Microsoft External Research for funding the NodeXL project at University of Maryland.

7. References

1. Fruchterman, T and Reingold, E: Graph Drawing by Force-directed Placement. Software – Practice And Experience, Vol. 21(1 1), 1129-1164 (November 1991)
2. Harel, D and Koren Y. A Fast Multi-Scale Algorithm for Drawing Large Graphs, Journal of Graph Algorithms and Applications. vol. 6, no. 3, pp. 179{202 (2002)
3. Hansen, D., Shneiderman, B. and Smith, M.: Analyzing Social Media Networks with NodeXL – Insight from a connected world, by Elsevier.
4. Perer, A and Shneiderman, B: Integrating Statistics and Visualization: Case Studies of Gaining Clarity during Exploratory Data Analysis, ACM SIGCHI Conference on Human Factors in Computing Systems (April 2008).
5. Brandes, Ulrik : A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, (2001)
6. Harish, Pawan and Narayanan,PJ: Accelerating large graph algorithms on the GPU using CUDA : Proceedings of the 14th international conference on High performance computing.
7. Dean, Jeffrey and Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.
8. Centrality – Wikipedia : http://en.wikipedia.org/wiki/Centrality
9. Hadoop : http://wiki.apache.org/hadoop/
10. Social Network Analysis http://en.wikipedia.org/wiki/Social_network#Social_network_analysis