Evaluating Centrality Metrics in Real-World Networks on GPU

Anuroop Sriram Kollu Gautham
Kishore Kothapalli P. J. Narayan R Govindarajulu
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India – 500 032.
Email:{anuroop@, gautham_k@} students.iiit.ac.in
{kkishore@, pjn@, gregeti@} iiit.ac.in

Abstract

GPGPU has received a lot of attention recently as a cost effective solution for high performance computing. In this paper we present a parallel algorithm for computing Betweenness centrality (BC) using CUDA. BC is an important metric in small world network analysis which is expensive to compute. While there are existing parallel implementations, ours is the first implementation on commodity hardware. Our algorithm exploits parallelism at multiple levels of granularity to achieve good performance. We conduct several experiments to show that the algorithm gives considerable speedup over sequential algorithms. We also provide a detailed analysis of the performance of the algorithm.

1. Introduction

Network analysis is currently an area of active research with applications ranging from social network analysis (friendship circles, organizational networks), phylogeny reconstruction and bio-informatics (protein interaction networks) to the Internet (web link analysis) etc. While these networks seem unrelated, empirical observations show that these graphs poses a number of similar properties like the small world effect (low average diameter), community structure, heavy tailed degree distributions etc.

Existing sequential algorithms have limited applicability given the size of many practical networks. While fast and scalable parallel algorithms have been proposed recently [1], [6], they have mostly targeted high-end architectures like the CRAY MTA-2. Of late, general purpose computing on graphics processors (GPGPU) has become a cost effective solution for high performance computing and there is a definite need for GPGPU versions of these network analysis algorithms.

In this paper, we present a parallel algorithm for evaluating both exact and approximate values for centrality metrics in networks using GPUs. These algorithms have been optimized for scale-free sparse graphs. To our knowledge, this is the first work on designing parallel algorithms for Social Network Analysis (SNA) for the GPU. Our algorithms have been designed to handle large graphs.

The paper is organized as follows: In section 2, we present the CUDA programming model; in section 3, we define Betweenness Centrality and give algorithms for it in sections 4 and 5. In section 6, we present our results and performance analysis, and finally conclude in section 7.

. Anuroop Sriram
. Kollu Gautham

2. CUDA Programming Model

The GPU is a massively multi-threaded processor containing hundreds of processing elements or cores, called the Scalar Processors (SPs). The SPs are arranged in groups of eight, called the Streaming Multiprocessors (SMs). These eight SPs execute in Single Instruction Multiple Thread (SIMT) fashion. Hence, all the SPs in an SM execute the same instruction at the same time[8].

The irregularity of memory accesses required for the current problem make it very hard to apply conventional GPU optimizations provided by CUDA. As the memory access pattern is not known in advance and there is very little data reuse in the algorithm, we cannot take advantage of the faster shared memory.

3. Betweenness Centrality

One of the fundamental problems in network analysis is to determine how important a given node/edge is relative to other nodes/edges in the network. For example, in a social network, one is interested in finding how important a person is. Quantifying centrality of nodes is a well studied problem and several metrics have been proposed for the same. The best metric to use is specific to the application domain and the network topology. The Betweenness Centrality (BC) [2] of a node signifies how important that node is in terms of communications between different nodes within the network assuming that communication takes place only through shortest paths.

Consider a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. Let the number of nodes be denoted by n and the number of edges be denoted by m. The BC of a node v is given by:

$$BC(v) = \sum_{s \neq t \neq v} \delta_{st}(v) \tag{1}$$

where $\delta_{st}(v)$ is the fraction of shortest paths from node s to node t which pass through the node v.

It can be observed that if many shortest paths pass through a node, then its BC is higher; also, if there exist many node disjoint shortest paths between a pair of nodes, then the contribution of each of these paths towards the BC of nodes that fall along these paths is small. Hence, a larger BC implies a well connected and central node.

In a similar fashion, we can also define the BC of an edge $v \rightarrow w$ as:

$$BC(v \rightarrow w) = \sum_{s \neq t \neq v} \delta_{st}(v \rightarrow w) \tag{2}$$

4. Sequential Algorithm

The best known sequential algorithm for computing the BC of all the nodes and edges in a graph is by Brandes [7] which needs a time of $O(n^2 + nm)$. So far, there is no algorithm for finding the BC of a single node which is asymptotically faster than finding the BCs of all the nodes. Brandes' algorithm is as follows:

The dependency of a node s on node v is defined as follows:

$$\delta_s(v) = \sum_t \delta_{st}(v) \tag{3}$$

Then it can be shown that:

$$\delta_s(v) = \sum_{w : v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \tag{4}$$

Here, σ_{sv} is the number of shortest paths between s and v, and $pred(s, w)$ is the set of predecessors of node w which fall on some shortest path to w from s. Similarly, the dependency of s on edge $v \rightarrow w$ is given by:

$$\delta_s(v \rightarrow w) = \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \tag{5}$$

Finally, we get

$$BC(v) = \sum_s \delta_s(v) \tag{6}$$

And

$$BC(v \rightarrow w) = \sum_s \delta_s(v \rightarrow w) \tag{7}$$

The BC of a node (or an edge) can be computed by finding the set of predecessors of that node on shortest paths from all other nodes. We can find the predecessors by using any shortest path algorithm. However, note that in a normal shortest path algorithm, the goal is just to find any shortest path to a node; in this case, however, we need to find and store every shortest path in the network (using the predecessors).

After finding the predecessors, we do a reverse of the shortest path algorithm, starting from nodes farthest from the start node and retracing the forward algorithm using the predecessor information. During this traversal, we find the dependencies of all the nodes and edges using Equations (4) through (7).

5. Parallel Implementation on CUDA

We need to find all the predecessors of all the nodes in the graph in shortest paths starting from each node in the graph. Hence, we would need $O(n^3)$ storage if we were to do find all the predecessor sets at once. Even in the best case, when we have only one predecessor for each node, we would still need $O(n^2)$ storage. Hence, an APSP strategy like the Floyd-Warshall's algorithm[10] does not work. Instead, we resort to performing breadth first forward and reverse traversals, with one start node at a time by reusing the memory.

During the forward traversal, we find the predecessors, level and the number of shortest paths to each of the nodes from a given start node. In the reverse traversal, we start at the nodes that were visited last and use the information found in the forward traversal to compute the dependencies of the predecessors. As we do not require storing the predecessor information after the reverse traversal has been done, we can reuse the storage if we perform only a few traversals at a time. To compute the shortest path, we can use the BFS algorithm for unweighted graphs or Dijkstra's SSSP algorithm for weighted graphs.

5.1. Algorithm

Algorithm 1 shows the pseudo code for unweighted graphs for one traversal. In each iteration of the forward traversal, we assign a set of nodes for each thread. Each thread first checks if its current node has already been visited. If yes, the thread proceeds to the next node; otherwise, it looks at the levels of its neighbors. For each of its neighbors which have already been visited, the node's level is updated and number of shortest paths to it is incremented by the number of shortest paths to that neighbour. Also, this neighbour is marked as a predecessor to this node. This process continues until there are no more updates.

Note that we could have implemented the forward traversal in two ways - each unvisited node updates its own level if any of its neighbors has already been visited, or each

Algorithm 1 Computing BC in CUDA (only the computation within a block is shown)

Input: $G(V, E)$, start node s
Output: Array $nodeBC[1..n]$ and $edgeBC[1..m]$

```
1:  currentLevel = 0, level[s] = 0, shortestPaths[s] = 1
2:  while there are updates do                    ▷ Forward Traversal
3:      for each node $n$ do in parallel
4:          if $level[n] == -1$ then
5:              for each visited neighbor $m$ of $n$ do
6:                  Add $m$ to predecessors of $n$
7:                  level[n] = currentLevel+1
8:                  $shortestPath[n] += shortestPath[m]$
9:              end for
10:         end if
11:     end for
12:     currentLevel++
13: end while
14: while $currentLevel > 0$ do                   ▷ Reverse Traversal
15:     for each node $n$ do in parallel
16:         if $level[n] == currentLevel$ then
17:             for each predecessor $p$ of $n$ do
18:                 Update dependency of $s$ on edge $p \to n$ using Equation (5)
19:             end for
20:         end if
21:     end for
22:     for each node $n$ do  in parallel
23:         if $level[n] == currentLevel - 1$ then
24:             for each neighbour $m$ of $n$ do
25:                 nodeDependency of s on $n$ += edgeDependency $p \to n$ on $s$
26:             end for
27:         end if
28:     end for
29:     currentLevel–
30: end while
```

already visited node updates all of its unvisited neighbors. We have chosen the former method because we can find the predecessors directly using the former method.

The reverse traversal proceeds in a similar manner as the forward traversal. In this, we start from the highest level and proceed towards the start node. In each iteration, the nodes of the 'current level' compute the dependencies of their predecessors using Equations (4) and (5).

5.2. Implementation details

The edges are stored as an array containing the adjacency lists of all the nodes. The edge array has a size m. The nodes are represented as an array of size n of indices into the edge array, such that the value at the i^{th} position points to the start of the edge list of the i^{th} node. The predecessors are represented as a boolean array of size m, such that the value at the i^{th} index is one if the i^{th} edge is from a predecessor to its successor. There is one predecessor array per traversal. Similarly, we have arrays of size n for the levels of nodes and the number of shortest paths to each node. An example undirected graph is shown in Figure 1.

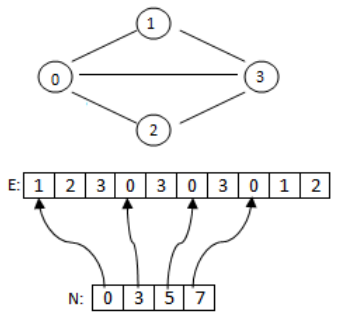

Figure 1. An example graph with 4 nodes and 5 edges. The corresponding nodes array (N) and edges array (E) are also shown.

We have chosen to perform one traversal (forward & backward) per block. Performing one traversal using more than one block requires returning back to the host at the end of each iteration as there is no means for synchronization across blocks in CUDA. This allowed us to extract the little inherent parallelism available in BFS while avoiding the expensive synchronization steps that make BFS on multiple SMs inappropriate for execution on CUDA. The algorithm shown in Algorithm 1 is run in each block with a different start node.

In line 18, we update the dependency on the edge from the predecessor to the current node using Equation (5). We could have updated the node dependency on the current node also in this line. But, the dependency on each edge (Equation (5)) is written to only once while the dependency on each node (Equation (4)) is updated as many times as there are successors for that node. Hence, there is a possibility for simultaneous writes to the same location while updating the dependencies of nodes, which can result in severe performance degradation.

To remove simultaneous writes, we modified the reverse traversal by updating only the dependencies on edges at this step. The dependencies on nodes of the predecessor level (currentLevel - 1) are calculated by adding up the dependencies on the edges from those nodes. This is done after all the nodes in the current level have been processed (lines 22-28). This calculation can be done in parallel for all the nodes. This process increases the amount of computation but eliminates the need for using expensive atomic operations. This change improved the performance significantly.

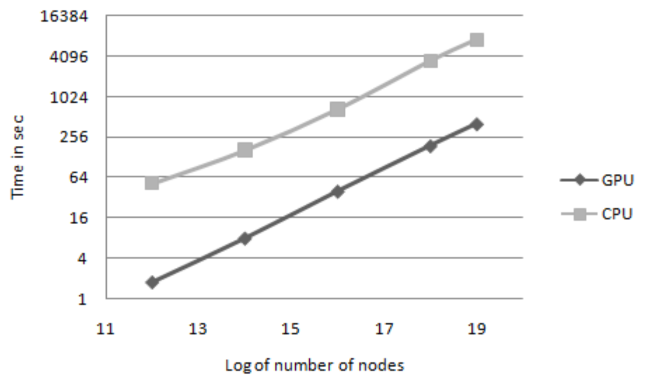

Figure 2. Runtime comparison of CPU and GPU on synthetic datasets of various sizes. The horizontal axis shows the scale of the graph. The number of nodes is 2^{scale} and the number of edges is $8 * 2^{scale}$.

Scale/dataset	Runtime on GPU (s)	Runtime on CPU (s)	Speedup
12	1.74	52.01	29.72
14	7.89	162.20	20.55
16	39.67	659.45	16.62
18	189.16	3441.87	18.20
19	402.33	7717.01	17.84
ND-Web	59.73	818.95	13.71

Table 1. Runtime on CPU and GPU in seconds for approximate BC with a sample of 2^{12} nodes. The number of nodes in the graph is 2^{scale} and the number of edges is $8 * 2^{scale}$. The last row shows the runtime on the ND-Web dataset.

5.3. Approximate BC Computation

For large-scale graphs, computation of exact BC is not computationally viable. Hence, we have also implemented an adaptive sampling based approximate BC algorithm proposed in [5]. This algorithm estimates the BC by sampling a subset of source nodes and performing the SSSP algorithm given in Algorithm 1 using only these. This algorithm is adaptive in the sense that the number of samples varies with the information obtained from each sample. The algorithm is based on the following theorem from [5]:

Theorem 1: For $0 < \epsilon < 0.5$, if the centrality of a vertex v is n^2/t for some constant $t \geq 1$, then with probability $\geq 1 - 2\epsilon$ its centrality can be estimated to within a factor of $1/\epsilon$ with ϵt samples of source vertices.

6. Results

This section summarizes our experimental results. We used an NVIDIA Tesla T10 processor which has 30 SMs with 8 SPs each. It has a global memory of 4 GB. As our storage requirements are O (m+n), we can process graphs with several million vertices and edges. We also implemented the sequential algorithm given in [7] on an AMD Athlon(tm) 64 X2 Dual Core Processor 4400+ at 3.0 GHz with 2 GB RAM. We used the ND-web dataset[4] to test our results. The ND-web is a network of over 0.32 million nodes representing web pages and 1.5 million edges representing links between them.

Further, we also tested our algorithm using synthetically generated small world networks generated using the Recursive MATrix (R-MAT) toolkit[9]. R-MAT is an algorithm to generate scale free graphs which works by recursively subdividing the adjacency matrix and distributing the edges into these partitions. We used undirected, unweighted versions of all the networks in our experiments.

We ran the approximate BC computation algorithm on ND-Web dataset and on the synthetic data using a sample of 2^{12} nodes on the GPU and the CPU. Our results are summarized in Figure2 and Table1. We obtained a speedup of 13.7 using on the ND-Web dataset and even higher speedups using the synthetic datasets.

6.1. Performance Analysis

In this section we provide a detailed performance analysis for the algorithm based on [11]. The GPU performance is limited by the following three characterstics of the BC algorithm:

1) *Non-contiguous memory access pattern*: As the degrees in a small world network are highly variable and the fact that adjacent neighbors generally do not constitute adjacent elements in the nodes array, we cannot make effective use of the available shared memory. This affects the performance significantly because the access time for the global memory is about 100-150 times the access time for the shared memory.
2) *Low arithmetic intensity*: The BC algorithm is highly memory intensive. In parallel programs, in general, memory access latency is hidden behind computations. However, the BC algorithm contains hardly any computation. This makes it very hard to hide the memory latency which causes in memory congestion. This leads to very low performance. To measure the effects of memory congestion, we plotted a graph between speedup vs number of SMs used (Figure 3). This graph shows that there is very little improvement in speedup between using 16 SMs and 30 SMs due to memory congestion.
3) *Unstructured parallelism*: In the BC algorithm, parallelism can be exploited at three levels of granularity: coarse-grained (running multiple traversals with different start nodes in parallel), medium-grained (processing different nodes of the same level in parallel), and fine-grained (processing the neighbors of the same node in parallel). Coarse-grained parallelism is embarrassingly parallel. However, multiple parallel traversals

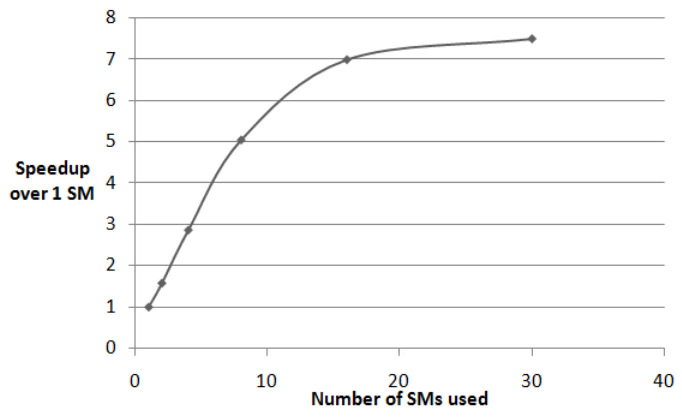

Figure 3. Speedup on GPU with different number of SMs with respect to one SM using the ND-Web dataset.

need more memory as we need to store more copies of the predecessor, level and shortest paths arrays. Hence, memory constraints limit the amount of coarse-grained parallelism that can be exploited.
Medium- and fine- grained parallelism depend on the degree of the nodes, which is highly variable. As the threads in a block run in a SIMT fashion, we cannot exploit a large amount of medium-grained parallelism because, if one thread finishes processing the neighbors of the node assigned to it, it has to wait till all the other threads in its block have also finished. Also, the degrees follow power laws, which means that most nodes have a very low degree. Hence, the fine-grained parallelism that can be exploited is also limited. Further, medium- and fine-grained parallelism add to memory congestion. As the algorithm is highly memory intensive, exploiting more parallelism at finer granularities may actually reduce the performance.

We exploit coarse-grained parallelism at the block level by running different traversals in different blocks and medium-grained parallelism at the thread level. As the algorithm already has a lot of memory congestion, exploiting fine-grained parallelism affects performance and hence, we refrain from exploiting it.

In addition to all these, the algorithm used for the GPU is also different from the one used for sequential processing. To determine the effect of the former on the performance, we implemented the GPU-like algorithm for sequential processing on the CPU and found that it ran 3 times slower than the queue-based sequential algorithm on the ND-Web dataset.

7. Conclusion and future work

In this paper we presented an algorithm for evaluating betweenness centrality of edges and nodes in small-world networks on the GPU. To our knowledge, this is the first attempt to implement a small world network analysis algorithm on the GPU. Our algorithms, though somewhat limited in speed by the large number of irregular memory accesses, gave considerable speedup over the CPU. We also analyzed the performance in detail and explained the reasons for the observed results. In the future, we plan to extend this algorithm to use multiple GPUs and also develop algorithms for other network analysis problems like community structure identification which use the betweenness centrality metric.

References

[1] D. A. Bader and K. Madduri, Parallel algorithms for evaluating centrality indices in real world networks, in emphThe 35th International Conference on Parallel Processing (ICPP 2006), 2006.

[2] L. C. Freeman, A set of measures of centrality based on betweenness, emphSociomtry, vol. 40, no. 1, pp. 3541, 1977.

[3] H. Jeong, S. P. Mason, A.-L. Barabasi, and Z. N. Oltvai, Lethality and centrality in protein networks, emphNature, vol. 411, p. 41, 2001.

[4] Notre Dame CNet resources. http://www.nd.edu/˜networks/ resources.htm

[5] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, Approximating Betweenness Centrality. *The 5th Workshop on Algorithms and Models for the Web-Graph (WAW2007)*, 2007

[6] D. A. Bader and K. Madduri, SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks, emphThe 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2008.

[7] U. Brandes, A faster algorithm for betweenness centrality, emphJournal of Mathematical Sociology, vol. 25, no. 2, pp. 163-177, 2001.

[8] *NVIDIA CUDA programming guide. http://www.nvidia.com/ object/cuda_develop.html*

[9] D. Chakraborthy, C. Faloustsos and Y. Zhang, Visualization of large networks with min-cut plots, A-plots and R-MAT. *International Journal of Human-Computer Studies*, vol. 65, pages 434-445, 2007.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 3 edition, emphMIT Press, 2009.

[11] D. Tu and G. Tan, Characterizing Betweenness Centrality Algorithm on Multi-core Architectures, emphIEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 182-189, 2009.