Computing Betweenness Centrality for Small World Networks on a GPU

Pushkar R. Pande David A. Bader
pushkar.pande@gatech.edu bader@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, 30332, USA

Abstract

Although a graphics processing unit (GPU) is a specialized device tailored primarily for compute-intensive, highly data-parallel computations; significant acceleration can be achieved on memory-intensive graph algorithms as well. In this work, we investigate the performance of a graph algorithm for computing vertex betweenness centrality for small world networks on 2 NVIDIA Tesla and Fermi GPUs and compare it to a parallel open source implementation on an Intel multicore CPU. For the test instances considered the betweenness computation on GPU was accelerated by as much as 19.68× compared to single thread CPU performance and more than 2× compared to multithread CPU performance using 16 OpenMP threads.

Introduction

Network analysis is an active area of research with applications in variety of domains such as social networks, protein interaction networks, computer security and disease spread. These real-world networks though highly unrelated display common features such as a small diameter, power law degree distribution and community structure, or in other words a small world topology. They are often very large in size with number of vertices and edges varying from millions to billions. These graphs are sparse and their analysis tends to be highly memory intensive. They have a large memory footprint, and a significant number of non-contiguous memory accesses to global data structures with low degree of spatial and temporal locality. Compared to other workloads most graph algorithms have little computation to hide latency to memory access.

Betweenness centrality is a key metric that is used to identify important actors in a network. It is a popular graph analysis technique based on shortest path enumeration. For a graph $G(V, E)$ with n vertices and m edges, the betweenness centrality of vertex $v \in V$ is defined as,

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

where σ_{st} denote the number of shortest paths between s and t, and $\sigma_{st}(v)$ denotes the number of shortest paths that pass through a specified vertex v. Let $\delta_{st}(v)$ denote the *pairwise dependency*, or the fraction of shortest paths between s and t that pass through v, i.e. $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. Then, the expression for betweenness centrality can be written as, $BC(v) = \sum_{s \neq v \neq t} \delta_{st}(v)$.

The NVIDIA C1060 (Tesla) graphics processor consists of 30 multiprocessors, each with 8 streaming processors (SM). The multiprocessor manages threads in groups of 32 parallel threads called *warps*. Each instruction is fetched and executed in parallel by the 8 SMs over 4 cycles for 32 data elements. The multiprocessors share an off-chip global memory. Accesses to global memory are not cached; hence it is important to realize coalesced memory access for good performance. Each multiprocessor has a software managed fast shared local buffer of 16 KB. The NVIDIA CUDA parallel programming model provides an abstraction of threads into a top level collection called grid and the grid is composed of multiple thread blocks.

Computing Betweenness

Traditional approach to compute betweenness centrality is to first calculate number and length of shortest paths between all pairs of vertices s and t, followed by *pairwise dependency* accumulation for each vertex v to obtain the betweenness centrality score. Using Floyd-Warshall algorithm for all-pairs shortest-paths problem augmented with path counting, results in an $O(n^3)$ algorithm that requires $O(n^2)$ space.

Brandes [3] presented a faster algorithm to exploit the sparse nature of small world networks that computes the betweenness centrality score for all vertices in $O(mn)$ time for unweighted graphs and requires $O(m + n)$ space. Let $\delta_s(v)$ be the dependency of source vertex s on vertex v defined as, $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$. The betweenness centrality of a vertex v can be then expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$. Brandes showed that $\delta_s(v)$ satisfies the following recursive relation,

$$\delta_s(v) = \sum_{w: d(s,w) = d(s,v) + 1} \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \delta_s(w)\right) \quad (2)$$

where $d(s, v)$ denotes the length of shortest path to v from source vertex s. The sequential algorithm computes betweenness in two stages as follows,

For each source vertex $s \in V$,

1. Perform augmented breadth-first traversal starting from source s to compute the length and number of shortest paths.
2. Revisit vertices starting from the farthest vertex from s and accumulate dependencies according to Eq. 2 to compute $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$

Parallelization

The first parallel algorithm and implementation for computing betweenness was given by Bader and Madduri in [1]. This targeted a massively multithreaded supercomputer, the Cray XMT. In [4], they presented a faster lock-free algorithm using successor multi-sets. Here we adapt this lock-free algorithm for GPU and evaluate its performance.

The betweenness centrality algorithm exhibits parallelism at three levels of granularity. At the coarse grain level, all the iterations from each source vertex s can be performed in parallel. Medium grain parallelism can be exploited by

processing in parallel all vertices that are in the same frontier i.e. equidistant from the source vertex. Further, one can choose to exploit fine grain parallelism by processing the neighbors of each vertex in parallel.

Coarse grained parallelism is limited by the amount of memory since multiple copies of data structures are required. Further, the centrality sum has to be updated atomically or accumulated via global reduction.

Since GPU offers a large number of threads and CUDA abstracts them into a hierarchy of threads grouped into grid and blocks, we chose to exploit parallelism at the medium and fine grain level by distributing vertices in the same frontier to different blocks in a grid. Each thread block then processes in parallel the neighbors of vertices assigned to it. During the dependency accumulation stage, vertices from each frontier starting from the farthest are processed in parallel by each thread according to Eq. 2.

Experimental Setup

The experiments were carried out on a number of synthetic unweighted – directed R-MAT graphs generated using the synthetic graph generator in SNAP [2]. More details on these graphs can be found in Table 1.

Graph	#Vertices	#Edges	Degree		#Connected Components
			Avg.	Max.	
syn1.gr	262,144	2,097,152	8	4,588	53,888
syn2.gr	524,288	4,194,304	8	4,063	115,376
syn3.gr	1,048,576	8,388,608	8	5,421	247,339
syn4.gr	1,000,000	10,000,000	10	10,030	536,229
syn5.gr	1,000,000	10,000,000	10	1,090	475,952

Table 1: R-MAT graphs used for the experiment

The GPUs used are an NVIDIA C1060 (Tesla) with a clock rate of 1.3 GHz, 30 × 8 cores, NVIDIA M2070 (Fermi), with a clock rate of 1.15 GHz, 14 × 32 cores. The multicore CPU has 2 quad-core Intel Xeon E5530 processors, with a clock rate of 2.4 GHz, hyper-threading enabled. The GPU implementation was compared to the open source OpenMP parallelized CPU implementation in SNAP. The computation was carried out for a subset of all-pairs shortest paths by using 500 source vertices.

Performance Evaluation and Optimizations

The runtime is dominated by the augmented breadth-first traversal stage. This stage does a considerable number of atomic updates to global data structures. Atomic operations on global memory are slow. Using fast shared memory for buffering writes to global memory helps realize coalescing and reduces number of atomic operations performed on the global data structures. For the test instances considered, the performance was improved by as much as 3× on GPUs with no cache (Tesla).

Due to power law degree distribution in small world networks, they often do not result in a balanced load partitioning among processing elements. Pre-processing the frontier for balanced load by distributing edges instead of vertices during the augmented breadth-first traversal improved the performance significantly. The adjacencies of each vertex in the frontier are split into *warp sized* chunks. The resulting chunks are then evenly distributed among the warps. Breadth-first traversals are performed a number of times, each starting from a different source vertex. It would be better to pre-process the graph just once to start with.

During the second stage of algorithm, assigning a warp instead of a thread per vertex to accumulate dependencies exploits the fine grain parallelism and provides a performance gain of up to 1.5 × by improving load balance.

Since a large number of vertices in a small world network have a low degree, there may not be enough work assigned per warp. Hence, we virtualized warp size to experiment with sizes of $4, 8, 16$ and 32.

Graph	CPU Time (s)		GPU Time (s)		Speed up on Fermi	
	$nt = 1$	$nt = 16$	Tesla	Fermi	$nt = 1$	$nt = 16$
syn1.gr	24.19	5.47	5.65	2.64	9.16	2.07
syn2.gr	80.72	15.78	13.48	6.33	12.75	2.49
syn3.gr	184.11	32.68	23.31	11.31	16.28	2.89
syn4.gr	93.63	17.35	13.77	6.11	15.32	2.84
syn5.gr	232.79	33.87	19.98	11.83	19.68	2.86

Table 2: GPU performance for betweenness compared to Intel Xeon with number of threads $nt = \{1, 16\}$ (hyper-threading enabled).

Table 2 summarizes the performance achieved on the manycore GPUs and the multicore CPU. The betweeness computation on Tesla is accelerated up to 11.65× compared to single thread CPU performance and 1.70× when compared to 16 threads (hyper-threading enabled). Though the GPU code was optimized for the Tesla architecture, running it *as is* on the Fermi GPU provided an *out of the box* performance gain of about 2× over Tesla. Compared to single thread performance and the best performance on Intel Xeon processor using 16 threads (hyper-threading enabled), the Fermi GPU performed up to 19.68× and 2.89× faster respectively.

References

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

[2] 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." *The 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2008*.

[3] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of mathematical sociology,* vol. 25, no. 2, pp. 163-177, 2001.

[4] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementation for Evaluating Betweenness Centrality on Massive Datasets", in *Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09), Rome, Italy, May 2009.*