DOI:10.1145/3230485

Accelerating GPU Betweenness Centrality

By Adam McLaughlin[a] and David A. Bader

Abstract

Graphs that model social networks, numerical simulations, and the structure of the Internet are enormous and cannot be manually inspected. A popular metric used to analyze these networks is Betweenness Centrality (BC), which has applications in community detection, power grid contingency analysis, and the study of the human brain. However, these analyses come with a high computational cost that prevents the examination of large graphs of interest.

Recently, the use of Graphics Processing Units (GPUs) has been promising for efficient processing of unstructured data sets. Prior GPU implementations of BC suffer from large local data structures and inefficient graph traversals that limit scalability and performance. Here we present a hybrid GPU implementation that provides good performance on graphs of arbitrary structure rather than just scale-free graphs as was done previously. Our methods achieve up to 13× speedup on high-diameter graphs and an average of 2.71× speedup overall compared to the best existing GPU algorithm. We also observe near linear speedup when running BC on 192 GPUs.

1. INTRODUCTION

Network analysis is a fundamental tool for domains as diverse as compilers,[17] social networks,[14] and computational biology.[5] Real world applications of these analyses involve tremendously large networks that cannot be inspected manually. An example of a graph analytic that has found significant attention in recent literature is BC. Betweenness centrality has been used for finding the best location of stores within cities,[20] studying the spread of AIDS in sexual networks,[13] power grid contingency analysis,[11] and community detection.[23] The variety of fields and applications in which this method of analysis has been employed shows that graph analytics require algorithmic techniques that make them performance portable to as many network structures as possible. Unfortunately, the fastest known algorithm for calculating BC scores has $O(mn)$ complexity for unweighted graphs with n vertices and m edges, making the analysis of large graphs challenging. Hence there is a need for robust, high-performance graph analytics that can be applied to a variety of network structures and sizes.

Graphics Processing Units (GPUs) provide excellent performance for regular, dense, and computationally demanding subroutines such as matrix multiplication. However, there has been recent success in accelerating irregular, memory-bound graph algorithms on GPUs as well.[6, 17, 19] Prior implementations of betweenness centrality on the GPU have outperformed their CPU counterparts, particularly on scale-free networks; however, they are limited in scalability to larger graph instances, use asymptotically inefficient algorithms that mitigate performance on high diameter graphs, and aren't general enough to be applied to the variety of domains that can leverage their results.

This article alleviates these problems by making the following contributions:

- We provide a work-efficient algorithm for betweenness centrality on the GPU that works especially well for networks with a large diameter.
- For generality, we propose an algorithm that chooses between leveraging either the memory bandwidth of the GPU or the asymptotic efficiency of the work being done based on the structure of the graph being processed. We present an online approach that uses a small amount of initial work from the algorithm to suggest which method of parallelism would be best for processing the remaining work.
- We implement our approach on a single GPU system, showing an average speedup of 2.71× across a variety of both real-world and synthetic graphs over the best previous GPU implementation. Additionally, our implementation attains near linear speedup on a cluster of 192 GPUs.

2. BACKGROUND

2.1. Definitions

Let a graph $G = (V, E)$ consist of a set V of $n = |V|$ vertices and a set E of $m = |E|$ edges. A path from a vertex u to a vertex v is any sequence of edges originating from u and terminating at v. Such a path is a *shortest path* if its sequence contains a minimal number of edges. A Breadth-First Search (BFS) explores vertices of a graph by starting a "source" (or "root") vertex and exploring its neighbors. The neighbors of these vertices are then explored and this process repeats until there are no remaining vertices to be explored. Each set of inspected neighbors is referred to as a *vertex frontier* and the set of outgoing edges from a vertex frontier is referred to as an *edge-frontier*. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices. A *scale-free* graph has a degree distribution that follows a power law, where a small number of vertices have a large number of outgoing

The original version of this paper is entitled "Scalable and High Performance Betweenness Centrality on the GPU" and was published in the *Proceedings of the 26th ACM/IEEE International Conference of High Performance Computing, Networking, Storage, and Analysis* (SC '14), 572–583.

[a]D.E. Shaw Research, New York, NY, USA.

edges and a large number of vertices have a small number of outgoing edges.[2] Finally, a *small world* graph has a diameter that is proportional to the logarithm of the number of vertices in the graph.[25] In these networks every vertex can be reached from every other vertex by traversing a small number of edges.

Representation of sparse graphs in memory. The most intuitive way to store a graph in memory is as an *adjacency matrix*. For unweighted graphs, element A_{ij} of the matrix is equal to 1 if an edge exists from i to j and is equal to 0 otherwise. The real-world graphs that we examine in this article, however, are *sparse*, meaning that a vast majority of the elements are zeros in the adjacency matrix representation of these data sets. Rather than using $O(n^2)$ space to store the entire adjacency matrix, we use the *Compressed Sparse Row* (CSR) format, as shown in Figure 1. This representation consists of two arrays: *row offsets* (`R`) and *column indices* (`C`). The column indices array is a concatenation of each vertex's adjacency list into an array of m elements. The row offsets array in an $n+1$ element array that points at where each vertex's adjacency list beings and ends within the column indices array. For example, the adjacency list of a vertex u starts at `C[R[u]]` and ends at `C[R[u+1]-1]` (inclusively).

2.2. Brandes's algorithm

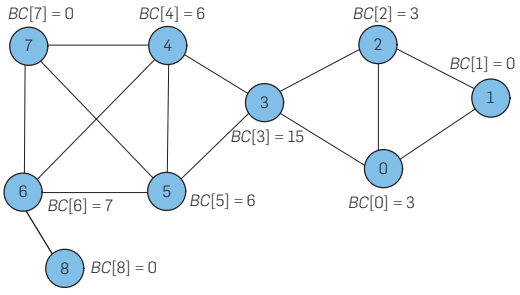
Betweenness centrality was originally developed in the social sciences for classifying people who were central to networks and could thus influence others by withholding information or altering it.[8] The metric attempts to distinguish the most influential vertices in a network by measuring the ratio of shortest paths passing through a particular vertex to the total number of shortest paths between all pairs of vertices. Intuitively, this ratio determines how well a vertex connects pairs of other vertices in the network. Formally, the Betweenness centrality of a vertex v is defined as:

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

where σ_{st} is the number of shortest paths between vertices s and t and $\sigma_{st}(v)$ is the number of those shortest paths that pass through v.

Consider Figure 1. Vertex 3 is the only vertex that lies on paths from its left (vertices 4 through 8) to its right (vertices 0 through 2). Hence vertex 3 lies on all of the shortest paths between these pairs of vertices and has a high BC score. In contrast, vertex 8 does not belong on a path between any pair of the remaining vertices in the graph and thus has a BC score of zero. Note that the scores reflected in Figure 1 treat a path from vertex u to vertex v as equivalent to a path from vertex v to vertex u since these paths are undirected. In other words, to avoid double counting the number of (undirected) shortest paths we divide the scores by two.

Figure 1. Example betweenness centrality scores and CSR representation for a small graph.



The magnitude of BC values also scales with the size of the network. For a fair comparison of BC values between vertices of two different graphs, a commonly used technique is to normalize the BC scores by their largest possible value[4]: $(n-1)(n-2)$. Such a comparison could be useful for comparing discrete slices of a network that changes over time.[15]

Naïve implementations of Betweenness Centrality solve the all-pairs shortest-paths problem using the $O(n^3)$ Floyd-Warshall algorithm and augment this result with path counting. Brandes improved upon this approach with an algorithm that runs in $O(mn)$ time for unweighted graphs.[3] The key concept of Brandes's approach is the *dependency* of a vertex v with respect to a given source vertex s:

$$\delta_s(v) = \sum_{w \in succ(v)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \quad (2)$$

The recursive relationship between the dependency of a vertex and the dependency of its successors allows a more asymptotically efficient calculation of the centrality metric. Brandes's algorithm splits the betweenness centrality calculation into two major steps:

1. Find the number of shortest paths between each pair of vertices.
2. Sum the dependencies for each vertex.

We can redefine the calculation of BC scores in terms of dependencies as follows:

$$BC(v) = \sum_{s \neq v} \delta_s(v) \quad (3)$$

2.3. GPU architecture and programming model

The relatively high memory bandwidth of GPUs compared to that of conventional CPUs has resulted in many high-performance GPU graph algorithms.[15, 17, 19] Compared to CPUs, GPUs tend to rely on latency hiding rather than caching and leverage a *Single-Instruction, Multiple-Thread* (SIMT) programming model. The SIMT model allows for transistors to be allocated to additional processor cores rather than structures for control flow management.

GPUs are comprised of a series of *Streaming Multiprocessors* (SMs), each of which manages hundreds of threads. The threads within each SM execute in groups of 32 threads (on current NVIDIA architectures) called *warps*. Although the execution paths of the threads within each warp may diverge, peak performance is attained when all threads within a warp execute the same instructions. Synchronization between the warps of a particular SM is inexpensive but properly synchronizing all of the SMs of the GPU requires the launch of a

separate *kernel*, or function that executes on the device. GPU threads have access to many registers (typically 255 or so), a small amount (typically 48KB) of programmer managed *shared memory* unique to each SM, and a larger *global memory* that can be accessed by all SMs.

3. PRIOR GPU IMPLEMENTATIONS

Two well-known GPU implementations of Brandes's algorithm have been published within the last few years. Jia et al.[10] compare two types of fine-grained parallelism, showing that one is preferable over the other because it exhibits better memory bandwidth on the GPU. Shi and Zhang present *GPU-FAN*[22] and report a slight speedup over Jia et al. by using a different distribution of threads to units of work. Both methods focus their optimizations on scale-free networks.

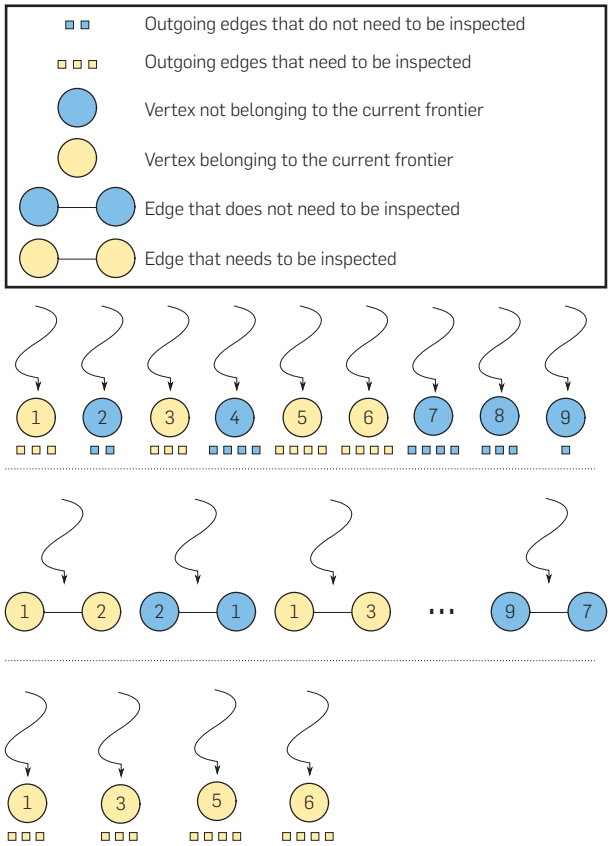
3.1. Vertex and edge parallelism

Jia et al. discussed two distributions of threads to graph entities: *vertex-parallel* and *edge-parallel*.[10] The vertex-parallel approach assigns a thread to each vertex of the graph and that thread traverses all of the outgoing edges from that vertex. In contrast, the edge-parallel approach assigns a thread to each edge of the graph and that thread traverses that edge only. In practice, the number of vertices and edges in a graph tend to be greater than the available number of threads so each thread sequentially processes multiple vertices or edges.

For both the shortest path calculation and the dependency accumulation stages the number of edges traversed per thread by the vertex-parallel approach depends on the out-degree of the vertex assigned to each thread. The difference in out-degrees between vertices causes a load imbalance between threads. For scale-free networks this load imbalance can be a tremendous issue, since the distribution of outdegrees follows a power law where a small number of vertices will have a substantial number of edges to traverse.[2] The edge-parallel approach solves this problem by assigning edges to threads directly. Both the vertex-parallel and edge-parallel approaches from Jia et al. use an inefficient $O(n^2 + m)$ graph traversal that checks if each vertex being processed belongs to the current depth of the search.

Figure 2 illustrates the distribution of threads to work for the vertex-parallel and edge-parallel methods. Using the same graph as shown in Figure 1, consider a Breadth-First Search starting at vertex 4. During the second iteration of the search, vertices 1, 3, 5, and 6 are in the vertex frontier, and hence their edges need to be inspected. The vertex-parallel method, shown in the top portion of Figure 2, distributes one thread to each vertex of the graph even though the edges connecting most of the vertices in the graph do not need to be traversed, resulting in wasted work. Also note that each thread is responsible for traversing a different number of edges (denoted by the small squares beneath each vertex), leading to workload imbalances. The edge-parallel method, shown in the middle portion of Figure 2, does not have the issue of load imbalance because each thread has one edge to traverse. However, this assignment of threads also results in wasted work because the edges that do not originate from vertices in the frontier do not need to be inspected in this particular iteration (but will be unnecessarily inspected during *every* iteration). Finally, the bottom portion of Figure 2 shows a work-efficient traversal iteration where each vertex in the frontier is assigned a thread. In this case only useful work is conducted although a load imbalance may exist among threads.

Figure 2. Illustration of the distribution of threads to units of work. Top: Vertex-parallel. Middle: Edge-parallel. Bottom: Work-efficient.



3.2. GPU-FAN

The *GPU-FAN* package from Shi and Zhang was designed for the analysis of biological networks representing protein communications or genetic interactions.[22] Similar to the implementation from Jia et al., GPU-FAN uses the edge-parallel method for load balancing across threads. The GPU-FAN package, however, focuses only on fine-grained parallelism, using all threads from all thread blocks to traverse edges in parallel for one source vertex of the BC computation at a time. In contrast, the implementation from Jia et al. uses the threads within a block traverse edges in parallel while separate thread blocks each focus on the independent roots of the BC computation.

4. METHODOLOGY

4.1. Work-efficient approach

Taking note of the issues mentioned in the previous section, we now present the basis for our work-efficient implementation of betweenness centrality on the GPU. Our approach leverages optimizations from the literature in addition to

our own novel techniques. The most important distinction between our approach and prior work is that we use explicit queues for graph traversal. Since levels of the graph are processed in parallel we use two queues to distinguish vertices that are in the current level of the search (Q_{curr}) from vertices that are to be processed during the next level of the search (Q_{next}). For the dependency accumulation stage we initialize S and its length. In this case, we need to keep track of vertices at all levels of the search and hence we only use one data structure to store these vertices. To distinguish the sections of S that correspond to each level of the search we use the *ends* array, where $ends_{len} = 1 + max_{v \in V} \{d[v]\}$ at the end of the traversal. Vertices corresponding to depth i of the traversal are located from index $ends[i]$ to index $ends[i+1] - 1$ (inclusively) of S. This usage of the *ends* and S arrays is analogous to the arrays used to store the graph in CSR format.

Algorithm 1: Work-efficient betweenness centrality shortest path calculation.

```
1  Stage 1: Shortest Path Calculation
2  while true do
3      for v ∈ Q_curr do in parallel
4          for w ∈ neighbors(v) do
5              if atomicCAS(d[w], ∞, d[v] + 1) = ∞ then
6                  t ← atomicAdd(Q_next_len, 1)
7                  Q_next[t] ← w
8              if d[w] = d[v] + 1 then
9                  atomicAdd(σ[w], σ[v])
10     barrier()
11     if Q_next_len = 0 then
12         depth ← d[S[S_len − 1]] − 1
13         break
14     else
15         for tid ← 0 ... Q_next_len − 1 do in parallel
16             Q_curr[tid] ← Q_next[tid]
17             S[tid + S_len] ← Q_next[tid]
18         barrier()
19         ends[ends_len] ← ends[ends_len − 1] + Q_next_len
20         ends_len ← ends_len + 1
21         Q_curr_len ← Q_next_len
22         S_len ← S_len + Q_next_len
23         Q_next_len ← 0
24         barrier()
```

A work-efficient shortest path calculation stage is shown in Algorithm 1. The queue Q_{curr} is initialized to contain only the source vertex. Iterations of the while loop correspond to the traversal of depths of the graph. The parallel for loop in Line 3 assigns one thread to each element in the queue such that edges from other portions of the graph aren't unnecessarily traversed. The atomic Compare And Swap (CAS) operation on Line 5 is used to prevent multiple insertions of the same vertex into Q_{next}. This restriction allows us to safely allocate $O(n)$ memory for Q_{next} instead of $O(m)$ in the case that duplicate queue entries are allowed. Since we only require one thread for each element in Q_{curr} rather than one thread for every vertex or edge in the graph, this atomic operation experiences limited contention and thus doesn't significantly reduce performance.

The conditional on Line 11 checks to see if the queue containing vertices for the next depth of the search is empty; if so, the search is complete, so we break from the outermost while loop. Otherwise, we transfer vertices from Q_{next} to Q_{curr}, add these vertices to the end of S for the dependency accumulation, and do the appropriate bookkeeping to set the lengths of these arrays.

Algorithm 2 shows a work-efficient dependency accumulation. We are able to eliminate the use of atomics by checking *successors* rather than the predecessors of each vertex. Rather than having multiple vertices that are currently being processed in parallel update the dependency of their common ancestor atomically, the ancestor can update itself based on its successors without the need for atomic operations.[14]

Algorithm 2: Work-efficient betweenness centrality dependency accumulation.

```
1  Stage 2: Dependency Accumulation
2  while depth > 0 do
3      for tid ← ends[depth] ... ends[depth + 1] − 1 do in
       parallel
4          w ← S[tid]
5          dsw ← 0
6          sw ← σ[w]
7          for v ∈ neighbors(w) do
8              if d[v] = d[w] + 1 then
9                  dsw ← dsw + (sw/σ[v])(1 + δ[v])
10         δ[w] ← dsw
11     barrier()
12     depth ← depth − 1
```

Note that the parallel for loop in Line 3 of Algorithm 2 assigns threads only to vertices that need to accumulate their dependency values; this is where the bookkeeping done to keep track of separate levels of the graph traversal in the *ends* array comes to fruition. Rather than naïvely assigning a thread to each vertex or edge and checking to see if that vertex or edge belongs to the current depth we instead can instantly extract vertices of that depth since they are a consecutive block of entries within S. This strategy again prevents unnecessary branch overhead and accesses to global memory that are made by previous implementations. For further implementation details we refer the reader to the associated conference paper.[16]

4.2. Rationale for hybrid methods

The major drawback of the approach outlined in the previous section is the potential for significant load imbalance between threads. Although our approach efficiently assigns

threads to units of useful work, the distribution of edges to threads is entirely dependent on the structure of the graph. Our approach is significantly faster than other methods on graphs with a large diameter because such graphs tend to have a more uniform distribution of outdegree. On scale-free or small world graphs, however, the algorithm outlined in the previous section does not improve performance. Based on this result we propose a hybrid approach that chooses between the edge-parallel and work-efficient methods based on the structure of the graph. Rather than preprocessing the graph to attempt to determine if it can be classified as a scale-free or small world graph, we implement our hybridization as an online approach.

Figure 3 illustrates our rationale behind the decision to use a hybrid algorithm. Each sub-figure shows how the vertex frontier evolves for three randomly chosen source vertices within a graph. Note that the axes of the sub-figures are on different scales to appropriately show trends in the frontiers. Although the position of the source vertex plays an important role in precisely how the vertex frontier changes with search iteration, we can see that the general sizes and changes in size of the vertex frontier across iterations of the search are more dependent on the overall structure of the graph. For high-diameter graphs such as *rgg_n_2_20* and *delaunay_n20* (Figures 3a and 3b), the vertex frontier grows gradually and is always a small portion of the total number of vertices in the graph. For graphs with a smaller diameter such as *kron_g500-logn20* (Figure 3c), the vertex frontier grows large after just a few iterations and contains over half of the total number of vertices in the graph at its peak.

Intuitively, for large vertex frontiers, the edge-parallel approach is favorable because of its memory throughput whereas for small vertex frontiers the work-efficient approach is favorable because the number of edges that will be traversed is significantly smaller than the total number of edges in the graph.

4.3. Sampling

The exact computation of betweenness centrality computes a BFS for each vertex in the graph. Since all of these searches are independent, they can be executed in parallel. For graphs whose vertices mostly belong to one large connected component, the amount of time to process each source vertex is roughly equivalent, as the same number of edges need to be traversed for each source vertex. Therefore the amount of time required to process k source vertices is roughly k times the time required to process one source vertex.[21]

Algorithm 3: Sampling method for selecting parallelization strategy.

Input: Set of n_{samps} connected component sizes (*keys*)
1 *sort*(*keys*)
2 *barrier*()
3 **if** $keys[n_{samps}/2] < \gamma * \log_2(n)$ **then**
4 //Switch to the edge-parallel method

Using the above analysis, an estimate of the average size of the connected components within the graph (and thus the preferred method of parallelism) is obtained by processing a small subset of its vertices. Algorithm 3 shows how this method is implemented. We initially use the work-efficient method to process a small subset of source vertices, recording the maximium depth of each of their BFS traversals. We then use the median of this set to be our estimate of the graph diameter. If this median is smaller than a threshold (determined by the parameter γ) then it is likely that our graph is a small-world or scale-free graph and that we should switch to using the edge-parallel approach.

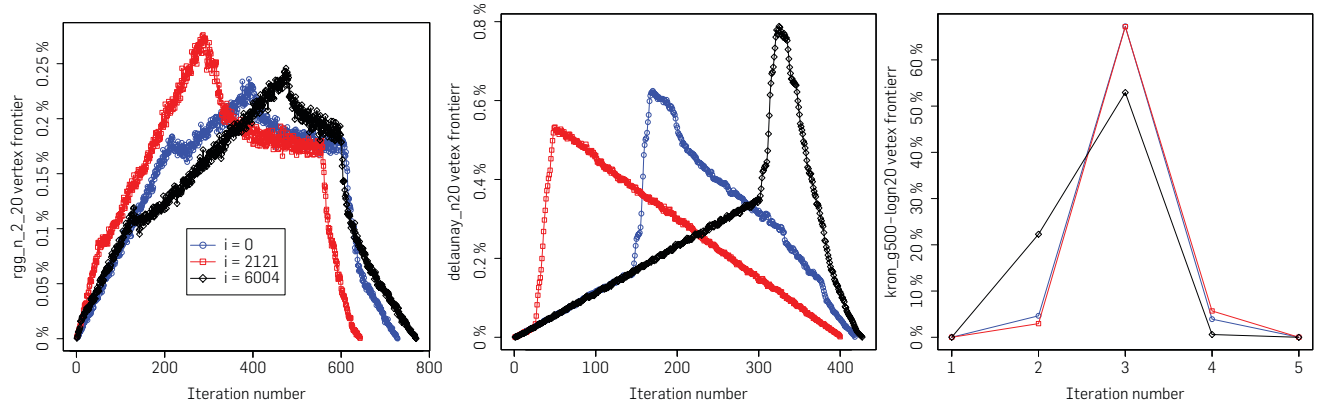
5. RESULTS

5.1. Experimental setup

Single-node GPU experiments were implemented using the Compute Unified Device Architecture (CUDA) 6.0 Toolkit. The CPU is an Intel Core i7-2600K processor running at 3.4 GHz with an 8MB cache and 16GB of DRAM. The GPU is a

Our implementation is available at https://github.com/Adam27X/hybrid_BC.

Figure 3. Evolution of vertex frontiers (as a percentage of total vertices) for different classifications of graphs.



(a) rgg_n_2_20 (b) delaunay_n20 (c) kron_g500-logn20

GeForce GTX Titan that has 14 SMs and a base clock of 837 MHz. The Titan has 6GB of GDDR5 memory and is a CUDA compute capability 3.5 ("Kepler") GPU.

Multi-node experiments were run on the Keeneland Initial Delivery System (KIDS).[24] KIDS has two Intel Xeon X5660 CPUs running at 2.8 GHz and three Tesla M2090 GPUs per node. Nodes are connected by an Infiniband Quadruple Data Rate (QDR) network. The Tesla M2090 has 16 SMs, a clock frequency of 1.3 GHz, 6GB of GDDR5 memory, and is a CUDA compute capability 2.0 ("Fermi") GPU.

We compare our techniques to both GPU-FAN[22] and Jia et al.[10] when possible, using their implementations that have been provided online. The graphs used for these comparisons are shown in Table 1. These graphs were taken from the 10th DIMACS Challenge,[1] the University of Florida Sparse Matrix Collection,[7] and the Stanford Network Analysis Platform (SNAP).[12] These benchmarks contain both real-world and randomly generated instances of graphs that correspond to a wide variety of practical applications and network structures. We focus our attention on the exact computation of BC, noting that our techniques can be trivially adjusted for approximation.

5.2. Scaling

First we compare how well our algorithm scales with graph size for three different types of graphs. Since the implementation of Jia et al. cannot read graphs that contain isolated vertices, we were unable to obtain results using this reference implementation for the random geometric (*rgg*) and simple Kronecker (*kron*) graphs. Additionally, since the higher scales caused GPU-FAN to run out of memory, we simply extrapolated what we would expect these results to look like from the results at lower scales (denoted by dotted lines). Note that from one scale to the next the number of vertices and number of edges both double.

Noting the log-log scale on the axes, we can see from Figure 4a that the sampling approach outperforms the algorithm from GPU-FAN by over 12× for all scales of *rgg*. It is interesting to note that the sampling approach only takes slightly more time than GPU-FAN when the sampling approach processes a graph four times as large. For the *delaunay* mesh graphs as shown in Figure 4b we can see that the edge-parallel method and the sampling approach both outperform GPU-FAN for all scales. The edge-parallel approach even outperforms the sampling approach for graphs containing less than 10,000 vertices; however, it should be noted that these differences in timings are trivial as they are on the order of milliseconds. As the graph size increases the sampling method clearly becomes dominant and the speedup it achieves grows with the scale of the graph. Finally, we compare the sampling approach to GPU-FAN for *kron* in Figure 4c. Although GPU-FAN is marginally faster than the sampling approach for the smallest scale graph we can see that the sampling approach is best at the next scale and the trend shows the amount by which the sampling approach is best grows with scale. Furthermore, neither of the previous implementations could support this type of graph at larger scales whereas the sampling method can support even larger scales.

5.3. Benchmarks

Figure 5 provides a comparison of the various parallelization methods discussed in this article to the edge-parallel method from Jia et al.[10] For road networks and meshes (*af_shell, del20, luxem*) all of the methods outperform the edge-parallel method by about 10×. The amount of unnecessary work performed by the edge-parallel method for these graphs is severe. For the remaining graphs (scale-free and small-world graphs) using the work-efficient method alone performs slower than the edge-parallel method whereas the sampling method is either the same or slightly better. In these cases we see the advantage of choosing our method of parallelization online.

In the most extreme case, the edge-parallel approach requires more than two and half days to process the *af_shell9* graph while the sampling approach cuts this time down to under five hours. Similarly, the edge-parallel approach takes over 48 min to process the *luxembourg.osm* road network whereas the sampling approach requires just 6 min. Overall, sampling performs 2.71× faster on average than the edge-parallel approach.

Table 1. Graph datasets used for this study.

Graph	Vertices	Edges	Max degree	Diameter	Description
af_shell9	504,855	8,542,010	39	497	Sheet metal forming
caidaRouterLevel	192,244	609,066	1,071	25	Internet router-level topology
cnr-2000	325,527	2,738,969	18,236	33	Web crawl
com-amazon	334,863	925,872	549	46	Amazon product co-purchasing
delaunay_n20	1,048,576	3,145,686	23	444	Random triangulation
kron_g500-logn20	1,048,576	44,619,402	131,503	6	Kronecker
loc-gowalla	196,591	1,900,654	29,460	15	Geosocial
luxembourg.osm	114,599	119,666	6	1,336	Road map
rgg_n_2_20	1,048,576	6,891,620	36	864	Random geometric
smallworld	100,000	499,998	17	9	Small world phenomenon

Figure 4. Scaling by problem size for three different types of graphs.

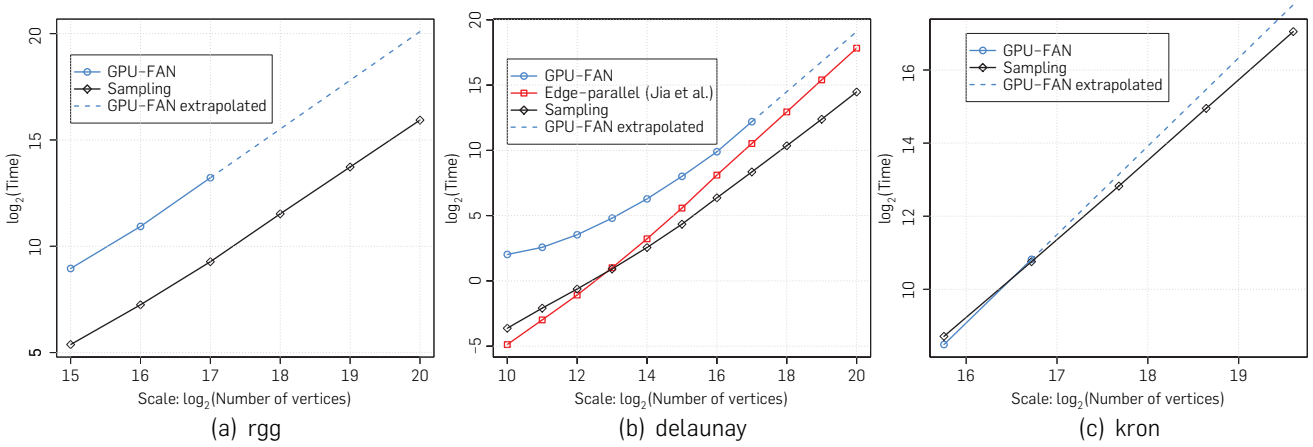
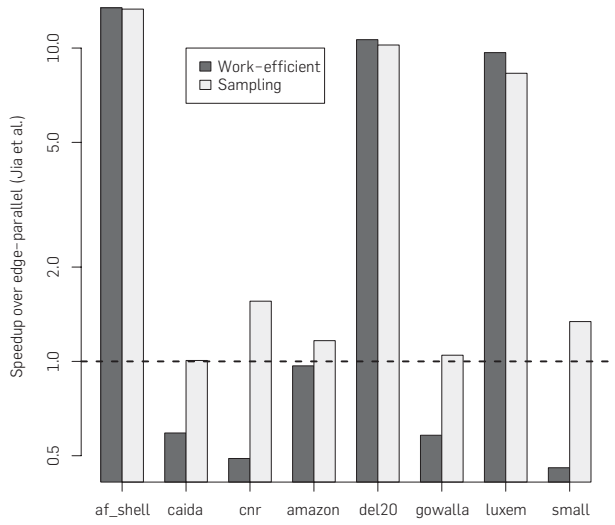


Figure 5. Comparison of work-efficient and sampling methods.



5.4. Multi-GPU experiments

Although our approaches leverage both coarse and fine-grained parallelism there is still more available parallelism than can be handled by a single GPU. Our methods easily extend to multiple GPUs as well as multiple nodes. We extend the algorithm by distributing a subset of roots to each GPU. Since each root can be processed independently in parallel, we should expect close to perfect scaling if each GPU has a sufficient (and an evenly distributed) amount of work.

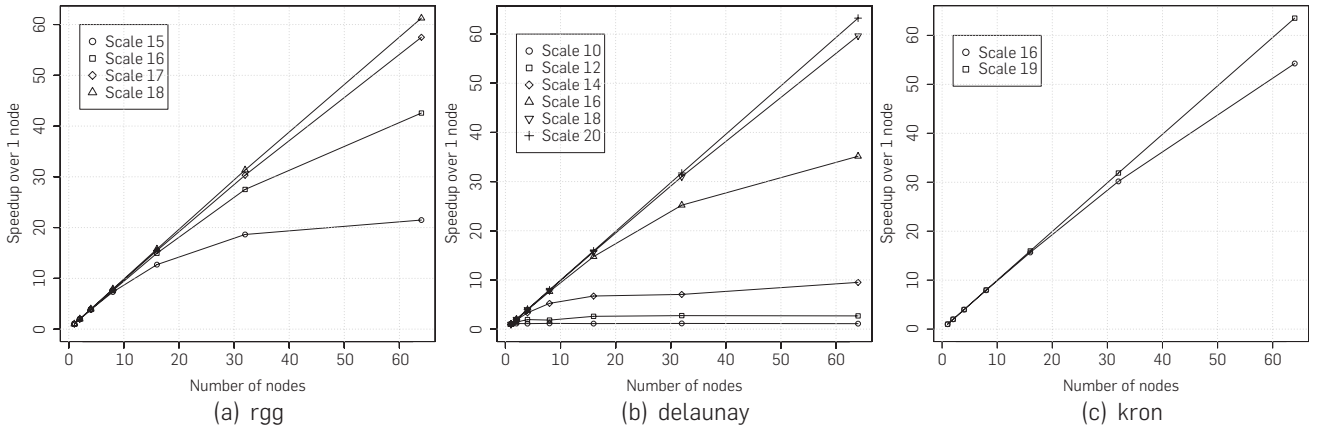
Since the local data structures for each root are independent (and thus only need to reside on one GPU), we replicate the data representing the graph itself across all GPUs to eliminate communication bottlenecks. Once each GPU has its local copy of the BC scores these local copies are accumulated for all of the GPUs on each node. Finally, the node-level scores are reduced into the global BC scores by a simple call to *MPI_Reduce*(). Figure 6 shows how well our algorithm scales out to multiple GPUs for *delaunay*, *rgg*, and *kron* graphs. It shows that linear speedup is easily achievable if the problem size is sufficiently large (i.e., if there is sufficient work for each GPU). Linear speedups are achievable at even smaller scales of graphs for denser network structures. For instance, using 64 nodes provides about a 35× speedup over a single node for scale 16 *delaunay* graph whereas using the same number of nodes at the same scale for *rgg* and *kron* graphs provides over 40× and 50× speedups respectively. The scaling behavior seen in Figure 6 is not unique to these graphs because of the vast amount of coarse-grained parallelism offered by the algorithm. For graphs of large enough size this scalability can be obtained independently of network structure.

6. CONCLUSION

In this article we have discussed various methods for computing Betweenness Centrality on the GPU. Leveraging information about the structure of the graph, we present several methods that choose between two methods of parallelism: edge-parallel and work-efficient. For high-diameter graphs using asymptotically optimal algorithms is paramount to obtaining good performance whereas for low-diameter graphs it is preferable to maximize memory throughput, even if unnecessary work is completed. In addition our methods are more scalable and general than existing implementations. Finally, we run our algorithm on a cluster of 192 GPUs, showing that speedup scales almost linearly with the number of GPUs, regardless of network structure. Overall, our single-GPU approaches perform 2.71× faster on average than the best previous GPU approach.

For future work we would like to efficiently map additional graph analytics to parallel architectures. The importance of robust, high-performance primitives cannot be overstated for the implementation of more complicated parallel algorithms. Ideally, GPU kernels should be modular and reusable; fortunately, packages such as Thrust[9] and CUB (CUDA Unbound)[18] are beginning to bridge this gap. A software environment in which users have access to a suite of high-performance graph analytics on the GPU

Figure 6. Multi-GPU scaling by number of nodes for various graph structures. Each node contains three GPUs.



would allow for fast network analysis and serve as a building block for more complicated programs.

Acknowledgments

The work depicted in this article was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." This work was also partially sponsored by NSF Grant ACI-1339745 (XScala). This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. Special thanks to Jeff Young for his assistance with running experiments on KIDS. Finally, we would also like to thank NVIDIA Corporation for their donation of a GeForce GTX Titan GPU. C

References

1. Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D. (eds) *Graph Partitioning and Graph Clustering—10th DIMACS Implementation Challenge*, volume 588 of *Contemporary Mathematics* (2013).
2. Barabási, A.-L., Albert, R. Emergence of scaling in random networks. *Science 286*, 5439 (1999), 509–512.
3. Brandes, U. A faster algorithm for betweenness centrality. *J. Math. Sociol. 25* (2001), 163–177.
4. Brandes, U. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks 30*, 2 (2008), 136–145.
5. Bullmore, E., Sporns, O. Complex brain networks: Graph theoretical analysis of structural and functional systems. *Nat. Rev. Neurosci. 10*, 3 (2009), 186–198.
6. Davidson, A.A., Baxter, S., Garland, M., Owens, J.D. Work-efficient parallel GPU methods for single-source shortest paths. In *International Parallel and Distributed Processing Symposium 28* (2014).
7. Davis, T.A., Hu, Y. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw. 38*, 1 (Dec. 2011), 1:1–1:25.
8. Freeman, L.C. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.
9. Hoberock, J., Bell, N. Thrust: A Parallel Template Library. Online at: http://thrust.googlecode. com, 42:43 (2010).
10. Jia, Y., Lu, V., Hoberock, J., Garland, M., Hart, J.C. Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems 2* (2011), 15–30.
11. Jin, S., Huang, Z., Chen, Y., Chavarria-Miranda, D., Feo, J., Wong, P.C. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (2010), 1–7.
12. Leskovec, J., Krevl, A. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
13. Liljeros, F., Edling, C.R., Amaral, L.A., Stanley, H.E., Aberg, Y. The web of human sexual contacts. *Nature 411*, 6840 (2001), 907–908.
14. Madduri, K., Ediger, D., Jiang, K., Bader, D., Chavarria-Miranda, D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (May 2009), 1–8.
15. McLaughlin, A., Bader, D. Revisiting edge and node parallelism for dynamic GPU graph analytics. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)* (2014).
16. McLaughlin, A., Bader, D.A. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis (SC)* (2014).
17. Mendez-Lojo, M., Burtscher, M., Pingali, K. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12 (New York, NY, USA, 2012). ACM, 107–116.
18. Merrill, D. CUDA Unbound. 2013. https://nvlabs.github.io/cub/ (accessed April 11, 2014).
19. Merrill, D., Garland, M., Grimshaw, A. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12 (New York, NY, USA, 2012). ACM, 117–128.
20. Porta, S., Latora, V., Wang, F., Strano, E., Cardillo, A., Scellato, S., Iacoviello, V., Messora, R. Street centrality and densities of retail and services in Bologna, Italy. *Environment and Planning B: Planning and design 36*, 3 (2009), 450–465.
21. Sariyüce, A.E., Saule, E., Kaya, K., Çatalyürek, Ü. Regularizing graph centrality computations. *J. Parallel Distrib. Comput. (JPDC)* (2014).
22. Shi, Z., Zhang, B. Fast network centrality analysis using GPUs. *BMC Bioinformatics 12*, 1 (2011), 149.
23. Soman, J., Narang, A. Fast community detection algorithm with GPUs and multicore architectures. In *International Parallel & Distributed Processing Symposium (IPDPS)* (IEEE, 2011), 568–579.
24. Vetter, J.S., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., Yalamanchili, S. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Comput. Sci. & Engineering 13*, 5 (2011), 90–95.
25. Watts, D.J., Strogatz, S.H. Collective dynamics of 'Small-World' networks. *Nature 393*, 6684 (1998), 440–442.

Adam McLaughlin (adam27x@gatech.edu), School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA.

David A. Bader (bader@cc.gatech.edu), School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA.