A GPU-based solution to fast calculation of betweenness centrality on large weighted networks (#19760)

First submission

Please read the **Important notes** below, the **Review guidance** on page 2 and our **Standout reviewing tips** on page 3. When ready **submit online**. The manuscript starts on page 4.

Important notes

Editor and deadline

John Owens / 3 Sep 2017

Files	4 Figure file(s) 1 Latex file(s) Please visit the overview page to **download and review** the files not included in this review PDF.
Declarations	No notable declarations are present

Review guidelines

Please read in full before you begin

How to review

When ready **submit your review online**. The review form is divided into 5 sections. Please consider these when composing your review:

1. **BASIC REPORTING**
2. **EXPERIMENTAL DESIGN**
3. **VALIDITY OF THE FINDINGS**
4. General comments
5. Confidential notes to the editor

You can also annotate this PDF and upload it as part of your review

To finish, enter your editorial recommendation (accept, revise or reject) and submit.

BASIC REPORTING

- Clear, unambiguous, professional English language used throughout.
- Intro & background to show context. Literature well referenced & relevant.
- Structure conforms to **PeerJ standards**, discipline norm, or improved for clarity.
- Figures are relevant, high quality, well labelled & described.
- Raw data supplied (see **PeerJ policy**).

EXPERIMENTAL DESIGN

- Original primary research within **Scope of the journal**.
- Research question well defined, relevant & meaningful. It is stated how the research fills an identified knowledge gap.
- Rigorous investigation performed to a high technical & ethical standard.
- Methods described with sufficient detail & information to replicate.

VALIDITY OF THE FINDINGS

- Impact and novelty not assessed. Negative/inconclusive results accepted. *Meaningful* replication encouraged where rationale & benefit to literature is clearly stated.
- Data is robust, statistically sound, & controlled.
- Conclusions are well stated, linked to original research question & limited to supporting results.
- Speculation is welcome, but should be identified as such.

The above is the editorial criteria summary. To view in full visit **https://peerj.com/about/editorial-criteria/**

7 Standout reviewing tips

The best reviewers use these techniques

Tip	*Example*
Support criticisms with evidence from the text or from other sources	*Smith et al (J of Methodology, 2005, V3, pp 123) have shown that the analysis you use in Lines 241-250 is not the most appropriate for this situation. Please explain why you used this method.*
Give specific suggestions on how to improve the manuscript	*Your introduction needs more detail. I suggest that you improve the description at lines 57- 86 to provide more justification for your study (specifically, you should expand upon the knowledge gap being filled).*
Comment on language and grammar issues	*The English language should be improved to ensure that your international audience can clearly understand your text. I suggest that you have a native English speaking colleague review your manuscript. Some examples where the language could be improved include lines 23, 77, 121, 128 – the current phrasing makes comprehension difficult.*
Organize by importance of the issues, and number your points	*1. Your most important issue* *2. The next most important item* *3. …* *4. The least important points*
Give specific suggestions on how to improve the manuscript	*Line 56: Note that experimental data on sprawling animals needs to be updated. Line 66: Please consider exchanging "modern" with "cursorial".*
Please provide constructive criticism, and avoid personal opinions	*I thank you for providing the raw data, however your supplemental files need more descriptive metadata identifiers to be useful to future readers. Although your results are compelling, the data analysis should be improved in the following ways: AA, BB, CC*
Comment on strengths (as well as weaknesses) of the manuscript	*I commend the authors for their extensive data set, compiled over many years of detailed fieldwork. In addition, the manuscript is clearly written in professional, unambiguous language. If there is a weakness, it is in the statistical analysis (as I have noted above) which should be improved upon before Acceptance.*

A GPU-based solution to fast calculation of betweenness centrality on large weighted networks

Rui Fan [1], **Ke Xu** [1], **Jichang Zhao** Corresp. [2]

[1] State Key Laboratory of Software Development Environment, Beihang University, Beijing, P. R. China
[2] School of Economics and Management, Beihang University, Beijing, P. R. China

Corresponding Author: Jichang Zhao
Email address: jichang@buaa.edu.cn

Recent decades have witnessed the tremendous development of network science, which indeed brings a new and insightful language to model real systems of different domains. Betweenness, a widely employed centrality in network science, is a decent proxy in investigating network loads and rankings. However, the extremely high computational cost greatly prevents its applicability on large networks. Though several parallel algorithms have been presented to reduce its calculation cost on unweighted networks, a fast solution for weighted networks, which are in fact more ubiquitous than unweighted ones in reality, is still missing. In this study, we develop an efficient parallel GPU-based approach to boost the calculation of betweenness centrality on quite large and weighted networks. Comprehensive and systematic evaluations on both synthetic and real-world networks demonstrate that our solution can arrive the performance of 3.5x to 6.5x speedup over the parallel CPU implementation by integrating the work-efficient and warp-centric strategies. Our algorithm is completely open-sourced and free to the community and it is public available through https://dx.doi.org/10.6084/m9.figshare.4542405. Considering the pervasive deployment and declining price of GPU on personal computers and servers, our solution will indeed offer unprecedented opportunities for exploring the betweenness related problems and spark followup works in network science.

A GPU-based solution to fast calculation of betweenness centrality on large weighted networks

Rui Fan[1], Ke Xu[1], and Jichang Zhao[2]

[1]**State Key Laboratory of Software Development Environment, Beihang University, Beijing, P. R. China**
[2]**School of Economics and Management, Beihang University, Beijing, P. R. China**

Corresponding author:
Jichang Zhao[2]

Email address: jichang@buaa.edu.cn

ABSTRACT

Recent decades have witnessed the tremendous development of network science, which ~~indeed~~ brings a new and insightful language to model real systems of different domains. Betweenness, a widely employed centrality in network science, is a decent proxy in investigating network loads and rankings. However, the extremely high computational cost greatly prevents its applicability on large networks. Though several parallel algorithms have been presented to reduce its calculation cost on unweighted networks, a fast solution for weighted networks, which ~~are in fact~~ more ubiquitous than unweighted ones in reality, is still missing. In this study, we develop an efficient parallel GPU-based approach to boost the calculation of betweenness centrality on quite large and weighted networks. Comprehensive and systematic evaluations on both synthetic and real-world networks demonstrate that our solution can arrive the performance of 3.5x to 6.5x speedup over the parallel CPU implementation by integrating the work-efficient and warp-centric strategies. Our algorithm is ~~completely~~ open-sourced and free to the community and it is public available through `https://dx.doi.org/10.6084/m9.figshare.4542405`. Considering the pervasive deployment and declining price of GPU on personal computers and servers, our solution will ~~indeed~~ offer unprecedented opportunities for exploring the betweenness related problems and spark followup works in network science.

INTRODUCTION

Being an emergent and multidisciplinary research area, the network science has attracted much efforts denoted from researchers of different backgrounds such as computer science, biology and physics in recent decades. In these contributions, betweenness centrality (BC) is always applied as a critical metric to measure nodes' or edges' significance (Ma and Sayama, 2015; Freeman, 1977; Barthelemy, 2004; Abedi and Gheisari, 2015; Goh et al., 2003). For example, Girvan and Newman developed a community detection algorithm based on edge betweenness centrality (Girvan and Newman, 2002), Leydesdorff applied centrality as an indicator of the interdisciplinarity of scientific journals (Leydesdorff, 2007) and Motter and Lai established a model of cascading failures with node load being its betweenness (Motter and Lai, 2002). However, the extremely high time and space complexity of calculating betweenness centrality greatly limits its applicability on large networks. Before the landmark work of Brandes (Brandes, 2001), the algorithm for computing betweenness centrality requires $O(n^3)$ time and $O(n^2)$ space. While Brandes reduced the complexity to $O(n+m)$ on space and $O(nm)$ and $O(nm+n^2\log n)$ on time for unweighted and weighted networks, respectively, where n is the number of vertices and m is the number of edges (Brandes, 2001). However, this improved algorithm still can not satisfy scientific computation requirements in the present information explosion era as more and more unexpected large networks emerge, such as online social networks, gene networks and collaboration networks. For example, Twitter possesses hundreds of millions active users which construct a huge online social network. However, a weighted network with one million nodes may take about one year to calculate its betweenness centrality

using Brandes' algorithm, which is an unbearable cost. Existing parallel CPU algorithms may reduce the time to several days, which is still too expensive. Because of this, there is a pressing need to develop faster BC algorithm for explorations of diverse domains.

General Purpose GPU (GPGPU) computing, which provides excellent parallelization, achieves higher performance compared to traditional CPU sequential algorithms in many issues including network science (Mitchell and Frank, 2017; Merrill et al., 2015; Wang et al., 2015; Harish and Narayanan, 2007; Cong and Bader, 2005). CUDA is the most popular GPU-computing framework developed by NVIDIA corporation and some researchers have even parallelized Brandes's algorithm by using it (Shi and Zhang, 2011; Sariyüce et al., 2013; McLaughlin and Bader, 2014). However, previous works concentrated on unweighted networks for simplification, but to our best knowledge, most realistic networks are weighted ones. The most significant difference of BC algorithm on unweighted and weighted networks is the shortest path segment. In weighted networks, Dijkstra algorithm should be used to solve the single source shortest path (SSSP) problem rather than Breadth First Search (BFS) algorithm. Many efforts in previous work have been devoted to the GPU version of SSSP problem using the well-known Dijkstra algorithm (Martín et al., 2009; Ortega-Arranz et al., 2013; Delling et al., 2011; Davidson et al., 2014). Although these algorithms have been presented and developed, establishing a parallel version of betweenness centrality algorithm on weighted networks is nontrivial because the original SSSP algorithm have to be modified in many critical points for this task and to our best knowledge, a proper and fast solution is still missing. Aiming at filling this vital gap, we propose a fast solution using CUDA to calculate BC on large weighted networks based on previous GPU BC algorithms and SSSP algorithms.

To make our algorithm more efficient, we make efforts to optimize it by employing several novel techniques to conquer the influence of irregular network structures. Real-world networks have many characters which could deteriorate the performance of GPU parallelization algorithms. For example, the frontier set of nodes is always small compared to the total number of vertices, especially for networks with great diameters. In the meantime, the majority of nodes do not need to be inspected in each step, hence processing all vertices simultaneously in traditional algorithms is wasteful. McLaughlin and Bader proposed a work-efficient strategy to overcome this problem (McLaughlin and Bader, 2014). Another well-known issue is that the power-law degree distribution in realistic networks brings about serious load-imbalance. Several methods were proposed in previous study to conquer this problem, e.g., Merrill et al. employed edge parallel strategy to avoid load-imbalance (Merrill et al., 2015) and Hong et al. dealt with this problem by using warp technique (Hong et al., 2011). In this paper, we systematically investigate the advantages and drawbacks of these previous methods and implement them in our algorithm to solve the above two problems. Experiments on both real-world and synthetic networks demonstrate that our algorithm outperforms the baseline GPU algorithm significantly. Our main contributions are listed as follows:

- Based on previous GPU parallel SSSP and betweenness centrality algorithms, we propose an efficient algorithm to calculate betweenness centrality on weighted networks, which achieves 3.5x to 6.5x speedup over the parallel CPU algorithm on realistic networks.
- We compare the traditional node-parallel method to the work-efficient version and the warp-centric method. Experiments on realistic networks and synthetic networks demonstrate that the combination of the two strategies works better than others, which achieves 2.55x average speedup over the baseline method on realistic networks.
- We package our algorithm to a useful tool which can be used to calculate both node and edge betweenness centrality on weighted networks. Researchers could apply this tool to conveniently calculate BC on weighted networks fast, especially on large networks. The source code is publicly available through `https://dx.doi.org/10.6084/m9.figshare.4542405`.

BACKGROUND

First we briefly introduce the ~~well-know~~ Brandes's algorithm and Dijkstra algorithm based on the preliminary definitions of network and betweenness centrality.

Brandes's algorithm

A graph can be defined as $G(V,E)$, where V is the set of vertices, and E is the set of edges. An edge is a node pair (u,v,w), which means that there is a link connecting nodes u and v, and its weight is

w. If the edge (u,v) exists, it can be traversed from u to v and from v to u because we only focus on undirected graphs in this paper. However, our algorithm can be expanded to directed graph version easily. A path $P=(s,...,t)$ is defined as a sequence of vertices connected by edges, where s is the starting node and t is the end node. The length of P is the sum of the weights of the edges involved in P. $d(s,t)$ is the distance between s and t, which represents the minimum length of all paths connecting s and t. σ_{st} denotes the number of shortest paths from s to t. According to the definition, we have $d(s,s)=0$, $\sigma_{ss}=1$, $d(s,t)=d(t,s)$ and $\sigma_{st}=\sigma_{ts}$ for undirected graph. $\sigma_{st}(v)$ denotes the number of shortest paths from s to t where v lies on. Based on these definitions, the betweenness centrality can be defined as

$$C_B(v)=\sum_{s\neq v\neq t\in V}\frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{1}$$

From the above definitions, the calculation of betweenness centrality can be naturally separated into the following two steps:

1. Compute $d(s,t)$ and σ_{st} for all node pairs (s,t),
2. Sum all pair-dependencies,

in which pair-dependency is defined as $\delta_{st}(v)=\frac{\sigma_{st}(v)}{\sigma_{st}}$. The first step consumes $O(mn)$ and $O(mn+n^2\log n)$ time for unweighted and weighted graph respectively, therefore the bottleneck of this algorithm is the second step, which requires $O(n^3)$ time. Brandes developed a more efficient BC algorithm which requires $O(mn)$ time for unweighted graph, and $O(mn+n^2\log n)$ time for weighted graph. The critical point is that the dependency of a node v when the source node is s is $\delta_s(v)=\sum_{u:v\in P_s(u)}\frac{\sigma_{sv}}{\sigma_{su}}(1+\delta_s(u))$. Applying this equation, we can accumulate the dependencies after computing the distance and number of shortest paths from a source vertex s to all other vertices, rather than after computing all pair shortest paths.

We can develop a parallel version based on Brandes's algorithm for unweighted graph because the graph is always traversed as a tree by using BFS algorithm. Given a source node s, the root of the tree is s and the tree produced by BFS method in the first step. In the second step, dependencies related to source node s are calculated from the bottom to the root of the tree and the nodes at the same level are isolated and have no influence to each other. As a result, the parallel version can explore nodes at the same level simultaneously in both of the two steps, which ~~will essentially boost the calculation~~.

Dijkstra algorithm

Dijkstra algorithm (Dijkstra, 1959) and Floyd-Warshall algorithm (Floyd, 1962) are commonly employed to solve shortest path problems. ~~While~~ Dijkstra algorithm is more adaptable to betweenness centrality problem because Brandes's algorithm accumulates dependencies after computing single source shortest paths (SSSP), rather than finding and storing all pair shortest paths. Dijkstra algorithm applies greedy strategy to solve SSSP. In this algorithm, the source node is s and if the shortest path from s and another node u is achieved, u will be settled. ~~According to be settled or not,~~ all nodes in graph G could be separated into two sets, which are settled vertices S and unsettled vertices U. An array D is used to store tentative distances from s to all nodes. At first, $D(s)=0$ and $D(u)=\infty$ for all other nodes. And the source node s is settled and considered as the frontier node to be explored. In the second step, for every node $u\in U$ and the adjacent frontier node f, if $D[f]+w(f,u)<D[u]$, $D[u]$ will be updated to $D[f]+w(f,u)$. Then the node $v\in U$ that has the smallest distance value will be settled and considered as the new frontier node and then the procedure goes back to the second step. The algorithm finishes when all nodes are settled. From the above description, Dijkstra algorithm has no parallel character as it picks one frontier node in each iteration. But this restriction can be loosed that several frontier vertices can be explored simultaneously which is similar to BFS parallel approach.

GPU-BASED ALGORITHM

Parallel betweenness centrality algorithm

In this section, we introduce the details of our GPU version BC algorithm on weighted graph. Firstly, we apply *Compressed Sparse Row* (CSR) format, which is widely used in graph algorithms, to store the

input graph (Bell and Garland, 2009; Davidson et al., 2014). It is space efficient that both of the vertex and edge consume one entry, and it is convenient to perform the traversal task on GPU. Moreover, edges related to the same vertex store consecutively in memory which makes warp-centric technique more efficient. For storing weighted graphs, another array that stores the weights of all edges is accordingly required.

We apply both coarse-grained (that one block processes one root vertex s) and fine-grained parallel (that threads within the block compute shortest paths and dependencies that related to s) strategies. The pseudo-code in this paper describes the parallel procedure of threads within a block. Algorithm 1 shows the initialization of required variables. U and F represent unsettled set and frontier set, respectively. v is unsettled if $U[v] = 1$ and is frontier node if $F[v] = 1$. d represents the tentative distance and $\sigma[v]$ is the number of shortest paths from s to v. $\delta[v]$ stores the dependencies of v. $lock$ stores locks for all nodes to avoid race condition. If the $lock[v] = 1$, changing the shortest path is not permited(see next section for detail). Vertices in the same level are recorded in S continuously and the start (or end) point in S of each level is stored in $ends$. In other words, S and $ends$ record the levels of traversal as CSR format and they are used in the dependency accumulation step. As can be seen in Algorithm 3, in the dependency accumulation part, we get nodes at the same level from S and $ends$ and accumulate dependencies of these nodes simultaneously. Note that in Algorithm 3 we only assign threads for nodes that need to be inspected rather than assign for all nodes, which enhances the efficiency by avoiding redundant threads. We update the dependency of edges at Line 12 in Algorithm 3 if edge betweenness is required.

Algorithm 1 Betweenness Centrality: Variable Initialization

1: **for** $v \in V$ **do in parallel**
2: $\quad U[v] \leftarrow 1$
3: $\quad F[v] \leftarrow 0$
4: $\quad d[v] \leftarrow \infty$
5: $\quad \sigma[v] \leftarrow 0$
6: $\quad \delta[v] \leftarrow 0$
7: $\quad lock[v] \leftarrow 0$
8: $\quad ends[v] \leftarrow 0$
9: $\quad S[v] \leftarrow 0$
10: **end for**
11: $d[s] \leftarrow 0$
12: $\sigma[s] \leftarrow 1$
13: $U[s] \leftarrow 0$
14: $F[s] \leftarrow 1$
15: $S[0] \leftarrow s; S_{len} \leftarrow 1$
16: $ends[0] \leftarrow 0; ends[1] \leftarrow 1; ends_{len} \leftarrow 2$
17: $\Delta \leftarrow 0$

Parallel Dijkstra algorithm

The parallel version of BFS procedure, which is applied in BC algorithm for unweighted network, could be modified naturally from its sequential version because vertices located at the same level of the BFS tree can be inspected simultaneously. And in the dependency accumulation step (step two), dependencies are calculated from low level vertices (nodes with largest depth in the tree) to the high level nodes (nodes that close to the source node) and nodes in the same level are calculated simultaneously. In the weighted version, the multi-level structure is also necessary in the dependency accumulation step to acquire parallelization. As can be seen in Fig 1(a), this structure should satisfy the condition $\forall u \in P_v, l_u < l_v$, where l_i means the level of node i in the multi-level structure and P_i represents the set of predecessors of vertex i. Previous high performance parallel SSSP algorithms such as Δ-stepping algorithm (Meyer and Sanders, 2003) only calculate the shortest path values, neglecting the number of shortest paths and the level relationships, which makes it inappropriate for our betweenness algorithm on weighted graphs. In this paper, we propose a variant of parallel Dijkstra algorithm, producing the number of shortest paths and the multi-level structure to fit our betweenness algorithm.

In the sequential Dijkstra algorithm, picking one frontier node each time makes its parallelization a

Algorithm 2 Betweenness Centrality: Shortest Path Calculation by Dijkstra Algorithm

```
while Δ < ∞ do
    for v ∈ V and F[v] = 1 do in parallel
        for w ∈ neighbors(v) do
            needlock ← true
            while needlock do
                if 0 = atomicCAS(lock[w], 0, 1) then
                    if U[w] = 1 and d[v] + weight_vw < d[w] then
                        d[w] ← d[v] + weight_vw
                        σ[w] ← 0
                    end if
                    if d[w] = d[v] + weight_vw then
                        σ[w] ← σ[w] + σ[v]
                    end if
                end if
                atomicExch(lock + w, 0)
                needlock ← false
            end while
        end for
    end for
    Δ ← ∞
    for v ∈ V do in parallel
        if U[v] = 1 and d[v] < ∞ then
            atomicMin(Δ, d[v] + Δ_node v)
        end if
    end for
    cnt ← 0
    for v ∈ V do in parallel
        F[v] ← 0
        if U[v] = 1 and d[v] < Δ then
            U[v] ← 0
            F[v] ← 1
            t ← atomicAdd(S_len, 1)
            S[t] ← v
            atomicAdd(cnt, 1)
        end if
    end for
    if cnt > 0 then
        ends[ends_len] ← ends[ends_len − 1] + cnt
        ends_len ← ends_len + 1
    end if
end while
```

Algorithm 3 Betweenness Centrality: Dependency Accumulation

```
depth ← ends_len − 1
while depth > 0 do
    start ← ends[depth − 1]
    end ← ends[depth] − 1
    for 0 ≤ i ≤ end − start do in parallel
        w ← S[start + i]
        dsw ← 0
        for v ∈ neighbors(w) do
            if d[v] = d[w] + weight_wv then
                c ← σ[w]/σ[v] ∗ (1 + δ[v])
                dsw ← dsw + c
                atomicAdd(edgeBC[w], c)
            end if
        end for
        δ[w] ← dsw
        if w ≠ s then
            atomicAdd(BC[w], δ[w])
        end if
    end for
    depth ← depth − 1
end while
```

difficult task. However, this restriction can be relaxed, which means that several nodes could be settled becoming frontier set and be inspected simultaneously in the next step. Moreover, these settled nodes satisfy the level-condition and because of this, they form a new level and will be inspected simultaneously in the dependency accumulation step. In this paper, we apply the method described in (Ortega-Arranz et al., 2013). In this method, $\Delta_{node\ v} = \min(w(v,u) : (v,u) \in E)$ is precomputed. Then we define Δ_i as

$$\Delta_i = \min\{(D(u) + \Delta_{node\ u}) : u \in U_i\}, \tag{2}$$

where $D(u)$ is the tentative distance of node u, U_i is the unsettled nodes set in iteration i. All nodes that satisfy the following condition

$$D(v) \leq \Delta_i \tag{3}$$

are settled and become frontier nodes. When applying Dijkstra algorithm in betweenness centrality calculation, the number of shortest paths should be counted and predecessor relationship between vertices in the same level is not permitted, otherwise the parallel algorithm will result in incorrect dependencies. To achieve this goal, the above condition should be modified to

$$D(v) < \Delta_i. \tag{4}$$

Fig. 1(b) demonstrates an example, in which vertex v_0 is the source node. If applying Eq. 3, v_1 and v_2 will be frontier nodes after inspecting v_0 in the first iteration, and the number of shortest paths will be 1 for both v_1 and v_2. Then v_1 and v_2 will be inspected simultaneously in next step. If processing v_2 first, the number of shortest paths for v_3 will be set to 1, while the correct value of shortest paths' number for v_3 should be 2. This mistake comes from the overambitious condition and v_2 should not be settled after the first iteration. Although the distance will be correct for all nodes using Eq. 3, but the number of shortest paths will be wrong. However, Eq. 4 will lead to correct shortest paths number for v_3 by only settling v_1 after first iteration. This condition could be found at Line 29 in Algorithm 2.

By performing Eq. 4 in SSSP step, we achieve correct shortest paths' number and we construct the multi-level structure by setting frontier nodes as a new level.

Algorithm 2 depicts our parallel Dijkstra algorithm in detail. The tentative distance and number of shortest paths are calculated which can be seen from Line 2 to Line 19. In this part, there will be a subtle parallel problem that several nodes in the frontier set may connect to the same node, as can be seen in Fig. 1(c). In this example, both v_1 and v_2 are in frontier set and connect to w, which results in the classical race condition problem. To avoid this situation, we define a lock for each node. The first thread focus on w will achieve the lock and other threads will not be permitted to change $d[w]$ and $\sigma[w]$. Note that other threads must not wait because in CUDA framework, a group of threads in a warp performs as a SIMD (Single Instruction Multiple Data) unit. Therefore, if w is not locked, current thread will achieve the lock and run the relax procedure. Otherwise it will run the circulation until another thread releases the lock. After computing d and σ for all nodes, we can achieve Δ_i based on the above analysis, as can be seen from Line 20 to Line 25. In the end, U, F, S and *ends* are updated for next iteration.

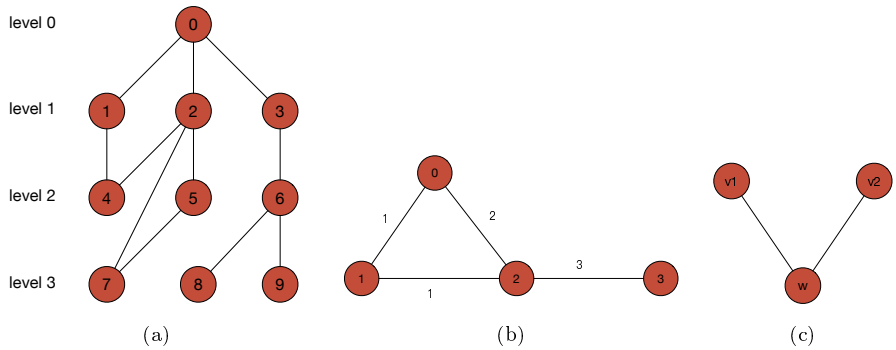

Figure 1. (a) An example of the multi-level structure. It is built in the SSSP step and will be used in the dependency accumulation step. Nodes in the same level are inspected simultaneously in both of the two steps. (b) An example of choosing frontier nodes, in which using Eq. 3 will make the number of shortest paths of v_3 incorrect. (c) An example of race condition. v_1 and v_2 are frontier nodes in one iteration, and both of which are connected with w.

Work-efficient method

As can be seen on Line 2 in Algorithm 2, threads will be assigned to all nodes but only nodes that in the frontier set will perform the calculation job, which may be inefficient. McLaughlin et al. figured out an excellent work-efficient technique to solve this problem (McLaughlin and Bader, 2014). In this paper, we develop our work-efficient version by absorbing this idea. F will be changed to a *queue* that stores all frontier nodes and a variable F_{len} is defined to recode the length of F, as can be seen in Algorithm 4. Then on Line 2 in Algorithm 5, threads can be assigned to $F[0] \sim F[F_{len}-1]$, which may be much smaller than the total number of nodes. At the same time, the method of updating F should also be changed, which can be seen in Algorithm 5.

Algorithm 4 Work-efficient betweenness Centrality: Variable Initialization

```
for v ∈ V do in parallel
    // initialize other variables except F
end for
F[0] ← s
F_len = 1
// initialize other variables
```

Warp-centric method

Many real-world networks have scale-free feature, which means their degree distributions follow power law. When implementing parallel graph algorithms through node parallel strategy, this feature brings about serious load-imbalance problem. Most nodes have low degrees while some nodes have extremely high degrees. Threads that assigned to high degree nodes will run slowly and other threads have to wait. Edge parallel strategy can solve this problem (Jia et al., 2011) but bring about other under-utilizations

Algorithm 5 Work-efficient betweenness Centrality: Shortest Path Calculation by Dijkstra Algorithm

```
while Δ < ∞ do
    for 0 ≤ i < F_len do in parallel
        v ← F[i]
        // inspect v
    end for
    // calculate Δ
    F_len ← 0
    for v ∈ V do in parallel
        if U[v] = 1 and d[v] < Δ then
            U[v] ← 0
            t ← atomicAdd(F_len, 1)
            F[t] ← v
        end if
    end for
    if F_len > 0 then
        ends[ends_len] ← ends[ends_len − 1] + F_len
        ends_len ← ends_len + 1
        for 0 ≤ i < F_len do in parallel
            S[S_len + i] ← F[i]
        end for
        S_len ← S_len + F_len
    end if
end while
```

at the same time. In this paper, we apply the novel warp-centric method (Hong et al., 2011), which allocates a warp rather than a thread to one node. Then threads within a warp focus on part of edges connected the specific node. As a result, each thread does less job for high degree nodes and the waiting time will be sharply decreased. Moreover, memory access patterns can be more coalesced than the conventional thread-level task allocation and because of this, the efficiency of memory access can be essentially improved.

Nevertheless, the warp-centric method also has some drawbacks. Firstly, node degree may be smaller than the warp size, which is always 32 in modern GPU. To solve this problem, Hong et al. proposed virtual warps (Hong et al., 2011). Secondly, the number of required threads will be raised as each node needs *WARP_SIZE* threads rather than one thread in this situation. But the number of threads in one block is fixed, hence each thread will be assigned to more nodes iteratively, which may result in low performance. We find that work-efficient method can relieve this problem because it requires less threads compared to the conventional node-parallel method, as can be seen in Fig. 2. In this paper, we apply the warp-centric method for both node-parallel and work-efficient method. As a result, we get four algorithms (see Tab. 2) that using different threads allocation strategies and we compare them on both real-world and synthetic networks.

EXPERIMENTS

Networks and settings

We collect seven weighted real-world networks from the Internet, which have broad types including collaboration networks, biological networks and social networks. They are publicly available in the Internet and have been analyzed extensively by previous literatures (Rossi and Ahmed, 2015; Bansal et al., 2007; Palla et al., 2008; Barabási and Albert, 1999; Leskovec and Krevl, 2014; De Domenico et al., 2013). The details of these networks are listed in Table 1. We develop a parallel CPU algorithm based on Graph-tool (Peixoto, 2014), which is an efficient network analysis tool whose core data and algorithms are implemented in C++ and supports parallel betweenness algorithm on weighted networks. We run our four GPU implementations on Geforce GTX 1080 using CUDA 8.0 Toolkit. The GeForce GTX 1080 is a compute capability 6.1 GPU designed under the Pascal architecture that has 20 multiprocessors, 8 GB of device memory, and a clock frequency of 1772 MHz. The CPU is Intel Core i7-7700K processor. The

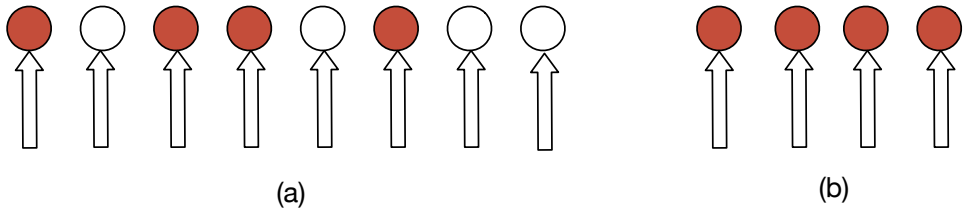

Figure 2. An example of threads allocation in node parallel method (a) and work-efficient method (b). Red nodes are frontier nodes that should be processed and an arrow represents a warp that be assigned to the corresponding node. Warp-centric method will waste more threads on nodes that do not need to be inspected. But combining warp-centric and work-efficient method can solve this problem, as shown in (b).

Core i7-7700K has a frequency of 4.2 GHz, 8 MB cache and eight physical processor cores. We use 4 threads since hyperthreading doesn't improve performance and we also run a sequential version because it is still widely applied by network researchers.

To further understand the effect of network structures to algorithms' performance, we generate two types of networks, which are Erdős-Rényi (ER) random graphs (Erdős and A., 1959) and Kronecker graphs (Leskovec et al., 2010). The degree distribution of ER random graph is Poisson, indicating its nodes' degrees are relatively balanced. While Kronecker graph possesses scale-free and small-world characters, which make it more similar to the realistic network. We uniformly assign random edge weights ranging from 1 to 10 as previous literature did (Martín et al., 2009; Ortega-Arranz et al., 2013).

Table 1. Details of networks from public dataset

Network	Vertices	Edges	Max degree	Average degree	Description
bio-human-gene1 (Rossi and Ahmed, 2015; Bansal et al., 2007)	22283	12345963	7940	1108.11	Human gene regulatory network
bio-human-gene2 (Rossi and Ahmed, 2015; Bansal et al., 2007)	14340	9041364	7230	1261.00	Human gene regulatory network
bio-mouse-gene (Rossi and Ahmed, 2015; Bansal et al., 2007)	45101	14506196	8033	643.28	Mouse gene regulatory network
ca-MathSciNet-dir (Rossi and Ahmed, 2015; Palla et al., 2008)	391529	873775	496	4.46	Co-authorship network
actors (Barabási and Albert, 1999)	382219	15038094	3956	78.69	Actors collaboration network
rt-higgs (Leskovec and Krevl, 2014; De Domenico et al., 2013)	425008	732827	31558	3.45	Twitter retweeting network
mt-higgs (Leskovec and Krevl, 2014; De Domenico et al., 2013)	116408	145774	11957	2.50	Twitter mention network

Results

From Tab. 2, we can see that all the GPU programs achieve better performance than both the sequential and parallel CPU version on all the seven real-world networks. The algorithm that applies work-efficient coupled with warp-centric technique is the best one for achieving 3.5x to 6.5x speedup compared to the parallel CPU method and 10x to 20x speedup compared to the sequential CPU algorithm respectively, and its performance could be essentially improved by assigning appropriate *WARP_SIZE*. Work-efficient method is more efficient than node-parallel in all networks, while warp-centric method performs better on large degree networks, such as the three biological networks. However, combining warp-centric method and work-efficient method always achieves better or approximately equal performance compared to work-efficient method alone because it needs less threads in each step, which accordingly relieves the influence of the second drawback of warp-centric method. For networks with low average degrees such as ca-MathSciNet-dir, rt-higgs and mt-higgs, applying warp-centric method with actual *WARP_SIZE* is always inefficient because nodes' degrees are always smaller than *WARP_SIZE*. Using smaller virtual *WARP_SIZE* performs better on these networks as shown in Tab. 2 and we will further demonstrate this later. By adjusting *WARP_SIZE* for low degree networks, the best performance program achieves 5x average speed-up compared to the parallel CPU implementation and 2.55x average speed-up compared to the baseline node-parallel strategy.

To deeply mining the relationship of the network structure and the performance of the four GPU implementations, we further run them on two types of synthetic graphs, as can be seen in Fig. 3. From Fig. 3(a), (b), (c) and (d), we find that work-efficient algorithm works better than node-parallel algorithm in all networks since it always reduces the required number of threads. As can be seen in Fig. 3(a)

Table 2. Benchmark results of the BC algorithms on weighted graphs, including a sequential CPU algorithm, a four threads CPU algorithm, NP (node-paralle), WE (work-efficient) and warp (warpx means the *WARP_SIZE* is x). Times are in seconds. The result of CPU sequential algorithm on actors network can not be provided because this program consumes too much time (more than one day).

Algorithm	bio-human-gene1	bio-human-gene2	bio-mouse-gene	ca-MathSciNet-dir	actors	rt-higgs	mt-higgs
CPU (sequential)	7494.09	3505.49	18300.83	49184.05	–	54717.96	1829.63
CPU (4 threads)	2245.61	1023.48	5460.26	21169.81	89196.19	21522.20	746.84
NP	1585.69	697.51	4407.42	6154.18	44137.50	4681.37	222.60
WE	1398.47	612.14	3742.69	4796.71	37803.60	4197.30	197.24
NP+warp32	511.73	196.67	1497.56	13883.50	32567.60	6205.65	337.89
WE+warp32	**403.51**	**159.68**	1214.86	4969.10	25382.20	4757.07	215.46
WE+warp4	784.86	327.93	1901.29	4593.97	28315.70	**3574.16**	**166.88**
WE+warp8	562.48	229.53	1365.80	**4579.23**	25469.50	3641.77	169.14
WE+warp16	439.58	174.51	**1170.26**	4706.05	**24715.40**	4008.30	184.45
best speed-up (over sequential CPU)	18.57	21.95	15.64	10.74	-	15.31	10.96
best speed-up (over parallel CPU)	5.57	6.41	4.67	4.62	3.61	6.02	4.48

and (b), warp-centric method works well on networks with large degrees, which is consistent with the conclusion in realistic networks. Note that for Kronecker graphs, warp-centric method works better than that for random graphs since Kronecker graphs have serious load-imbalance problem and warp-centric technique appropriately solves it. While for ER random graphs in Fig. 3(a), the advantage of warp-centric method is only the efficient memory access. For low degree graphs, warp-centric method works even worse than node-parallel strategy because the degrees are always smaller than *WARP_SIZE*, as can be seen in Fig. 3(c) and (d). For random graphs, the performance of warp-centric method is extremely poor when the average degree is smaller than 8 and Fig. 3(e) explains the reason. The small average degree brings about large average depth, which means that the average size of the frontier sets is small. In this case, warp-centric method assigns more useless threads to nodes that do not need inspections. On the contrary, as the degree grows, it is closer to *WARP_SIZE* and the depths drop down sharply, which make the warp-centric method performs much better. While low-degree Kronecker graphs have power-law degree distributions and small average depths, which make warp-centric method works not as bad as on random graphs. However, the combination of the two methods always runs faster than applying work-efficient method alone because it avoids the second drawback of warp-centric method, which is discussed in the previous section. In conclusion, work-efficient method always achieves better performance while the performance of warp-centric method relies on networks' structures but the joint version always achieves the best performance.

From the above analysis, applying smaller *WARP_SIZE* may accelerate the two implementations which using warp-centric method when the networks' average degree is small. And this hypothesis can be verified in Fig. 4. We apply smaller *WARP_SIZE* on rt-higgs network, mt-higgs network and other two synthetic graphs whose average degrees are both four. From Fig. 4(a) and (b), we find that implementations with smaller *WARP_SIZE* ~~do~~ perform better than both of the baseline node-parallel algorithms and the large *WARP_SIZE* algorithm on both of the low-degree realistic networks. And when coupled with work-efficient method, algorithms with smaller *WARP_SIZE* also perform better than both of the work-efficient strategy alone and the combination of work-efficient and large *WARP_SIZE*. The reason is that small *WARP_SIZE* reduces the required number of threads and then eliminates the waste of assigning more threads to a node than its degree. The implementations which have small *WARP_SIZE* and coupled with work-efficient method achieve the best performance because they avoid both drawbacks of warp-centric method but utilize the advantages of this technique. The results on low-degree Kronecker graph is similar as on realistic networks since Kronecker graph is similar with real-world network. For ER random graphs, the algorithm with small *WARP_SIZE* does not achieve better performance compared to node-parallel version because the large average depth, which is analyzed in previous section. However, when coupled with work-efficient method, the implementations with small *WARP_SIZE* perform slightly better than the work-efficient algorithm, which further proves the excellence and stability of the joint algorithm. In summary, the joint algorithm are most efficient and insensitive to network structure. And if we choose an appropriate *WARP_SIZE*, its performance could be even better.

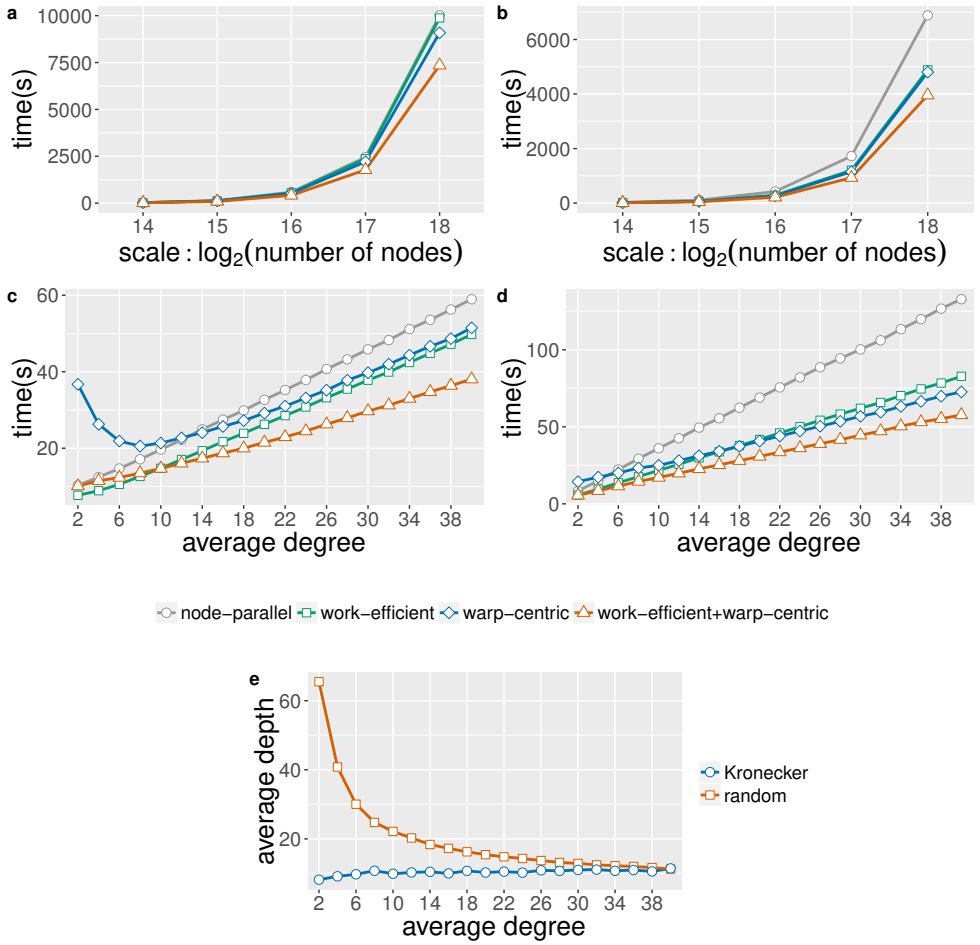

Figure 3. Performance of the four implementations on ER random and Kronecker graphs. The *WARP_SIZE* is fixed to 32 in the two warp-centric methods. (a) and (b) tune the number of nodes from 2^{14} to 2^{18} for ER random and Kronecker graphs, respectively. And the average degrees are fixed to 32 for both of the two types of networks. (c) and (d) separately tune the average degrees for random and Kronecker networks, in which the random networks have 20,000 vertices and the Kronecker networks have 2^{15} nodes. (e) illustrates the average depths of search trees for random graphs used in (c) and Kronecker graphs used in (d). Networks with larger depths have smaller average frontier sets, indicating the poor performance with parallelism.

CONCLUSION

Existing GPU version of betweenness centrality algorithms only concentrate on unweighted networks for simplification. Our work that computing betweenness centrality on large weighted networks bridges this gap and achieves prominent efficiency enhancement compared to the CPU implementation. Moreover, we apply two excellent techniques which are work-efficient and warp-centric methods in our algorithm. Work-efficient method allocates threads more efficiently and warp-centric method solves the load imbalance problem and simultaneously optimizes the memory access. We compare these implementations with sequential and parallel CPU algorithm in realistic networks. The results show that GPU parallel algorithms perform much better than the CPU algorithms and the algorithm which integrates the two techniques is the best, achieving 3.5x to 6.5x speedup over the parallel CPU version and 10x to 20x speedup over the sequential CPU version respectively. Results on synthetic random graphs and Kronecher graphs further justify the outperformance of our solution.

For future work, we will consider implementing GPU algorithm to process dynamic networks. When networks changes a little (like few new nodes come in or several links vanish), calculating betweenness

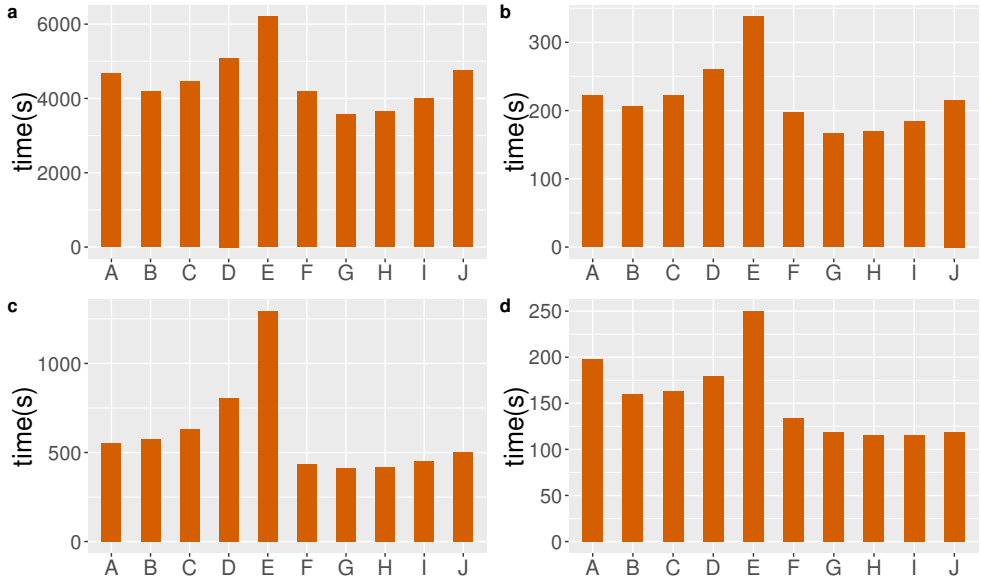

Figure 4. Applying other *WARP_SIZE* on several low-degree networks. A-J represent node-parallel, warp4, warp8, warp16, warp32, work-efficient, work-efficient+warp4, work-efficient+warp8, work-efficient+warp16, work-efficient+warp32, respectively. (a) and (b) are rt-higgs network and mt-higgs network, respectively, on which smaller warp size achieves better performance than both node-parallel method and the algorithm with large *WARP_SIZE*. (c) is a random graph with 2^{17} nodes whose average degree is four. Warp-centric method can not accelerate the speed when combining node-parallel strategy. But when combining small *WARP_SIZE* with work-efficient method, the performance will be slightly better than applying work-efficient method alone. (d) is Kronecker graph with 2^{17} nodes and the average degree is four, on which smaller *WARP_SIZE* achieves better performance.

centrality for all nodes is unnecessary because betweenness centrality of most nodes and edges will not be changed. Some previous works have explored the sequential algorithm on this issue (Lee et al., 2016; Singh et al., 2015; Nasre et al., 2014). We plan to develop GPU version of these algorithms to achieve better performance.

ACKNOWLEDGMENTS

This work was supported by NSFC (Grant No. 71501005) and the fund of the State Key Lab of Software Development Environment (Grant Nos. SKLSDE-2015ZX-05 and SKLSDE-2015ZX-28). R. F. also thanks the Innovation Foundation of BUAA for PhD Graduates.

REFERENCES

Abedi, M. and Gheisari, Y. (2015). Nodes with high centrality in protein interaction networks are responsible for driving signaling pathways in diabetic nephropathy. *PeerJ*, 3:e1284.

Bansal, M., Belcastro, V., Ambesi-Impiombato, A., and Di Bernardo, D. (2007). How to infer gene networks from expression profiles. *Molecular systems biology*, 3(1).

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.

Barthelemy, M. (2004). Betweenness centrality in large complex networks. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):163–168.

Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 18:1–18:11.

Brandes, U. (2001). A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177.

Cong, G. and Bader, D. A. (2005). An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps). In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05, pages 45b–45b, Washington, DC, USA. IEEE Computer Society.

Davidson, A., Baxter, S., Garland, M., and Owens, J. D. (2014). Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE Computer Society.

De Domenico, M., Lima, A., Mougel, P., and Musolesi, M. (2013). The anatomy of a scientific rumor. *Scientific reports*, 3.

Delling, D., Goldberg, A. V., Nowatzyk, A., and Werneck, R. F. (2011). Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, pages 921–931. IEEE Computer Society.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271.

Erdős, P. and A., R. (1959). On random graphs. *Publicationes Mathematicae*, 6:290–297.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM*, 5(6).

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41.

Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826.

Goh, K.-I., Oh, E., Kahng, B., and Kim, D. (2003). Betweenness centrality correlation in social networks. *Phys. Rev. E*, 67:017101.

Harish, P. and Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg. Springer-Verlag.

Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. (2011). Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 267–276.

Jia, Y., Lu, V., Hoberock, J., Garland, M., and Hart, J. C. (2011). Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems*, 2:15–30.

Lee, M.-J., Choi, S., and Chung, C.-W. (2016). Efficient algorithms for updating betweenness centrality in fully dynamic graphs. *Inf. Sci.*, 326(C):278–296.

Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., and Ghahramani, Z. (2010). Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Leydesdorff, L. (2007). Betweenness centrality as an indicator of the interdisciplinarity of scientific journals. *Journal of the American Society for Information Science and Technology*, 58(9):1303–1319.

Ma, X. and Sayama, H. (2015). Mental disorder recovery correlated with centralities and interactions on an online social network. *PeerJ*, 3:e1163.

Martín, P. J., Torres, R., and Gavilanes, A. (2009). Cuda solutions for the sssp problem. In *Proceedings of the 9th International Conference on Computational Science*, pages 904–913. Springer-Verlag.

McLaughlin, A. and Bader, D. A. (2014). Scalable and high performance betweenness centrality on the gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 572–583.

Merrill, D., Garland, M., and Grimshaw, A. (2015). High-performance and scalable gpu graph traversal. *ACM Trans. Parallel Comput.*, 1(2).

Meyer, U. and Sanders, P. (2003). Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114 – 152. 1998 European Symposium on Algorithms.

Mitchell, R. and Frank, E. (2017). Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3:e127.

Motter, A. E. and Lai, Y.-C. (2002). Cascade-based attacks on complex networks. *Phys. Rev. E*, 66:065102.

Nasre, M., Pontecorvi, M., and Ramachandran, V. (2014). Betweenness centrality–incremental and faster. In *International Symposium on Mathematical Foundations of Computer Science*, pages 577–

588. Springer.

Ortega-Arranz, H., Torres, Y., Llanos, D. R., and Gonzalez-Escribano, A. (2013). A new gpu-based approach to the shortest path problem. In *High Performance Computing and Simulation*, pages 505–511.

Palla, G., Farkas, I. J., Pollner, P., Derényi, I., and Vicsek, T. (2008). Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, 10(12):123026.

Peixoto, T. P. (2014). The graph-tool python library. *figshare*.

Rossi, R. A. and Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Sariyüce, A. E., Kaya, K., Saule, E., and Çatalyürek, Ü. V. (2013). Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85.

Shi, Z. and Zhang, B. (2011). Fast network centrality analysis using gpus. *BMC Bioinformatics*, 12:149.

Singh, R. R., Goel, K., Iyengar, S., and Gupta, S. (2015). A faster algorithm to update betweenness centrality after node alteration. *Internet Mathematics*, 11(4-5):403–420.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2015). Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, New York, NY, USA. ACM.