Betweenness Centrality on GPUs and Heterogeneous Architectures

Ahmet Erdem Sarıyüce[1,2], Kamer Kaya[1], Erik Saule[1], Ümit V. Çatalyürek[1,3]

Depts. [1]Biomedical Informatics, [2]Computer Science and Engineering, [3]Electrical and Computer Engineering
The Ohio State University
Email:{*aerdem,esaule,kamer,umit*}*@bmi.osu.edu*

ABSTRACT

The betweenness centrality metric has always been intriguing for graph analyses and used in various applications. Yet, it is one of the most computationally expensive kernels in graph mining. In this work, we investigate a set of techniques to make the betweenness centrality computations faster on GPUs as well as on heterogeneous CPU/GPU architectures. Our techniques are based on virtualization of the vertices with high degree, strided access to adjacency lists, removal of the vertices with degree 1, and graph ordering. By combining these techniques within a fine-grain parallelism, we reduced the computation time on GPUs significantly for a set of social networks. On CPUs, which can usually have access to a large amount of memory, we used a coarse-grain parallelism. We showed that heterogeneous computing, i.e., using both architectures at the same time, is a promising solution for betweenness centrality. Experimental results show that the proposed techniques can be a great arsenal to reduce the centrality computation time for networks. In particular, it reduces the computation time of a 234 million edges graph from more than 4 months to less than 12 days.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; C.1.2 [**Processor Architectures**]: Multiprocessors—*Parallel processors*]; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms

Algorithms, Performance

Keywords

Betweenness, heterogeneous computing, shared memory parallelism, GPUs, virtual vertices, graph compression

GPGPU-6 Workshop '13 Houston, TX, USA

1. INTRODUCTION

Centrality metrics, such as betweenness and closeness, quantify how central a node is in a network. They have been used successfully to carry analyses for various purposes such as structural analysis of knowledge networks [16, 19], power grid contingency analysis [8], quantifying importance in social networks [13], analysis of covert networks [10] and decision/action networks [4], and even for finding the best store locations in cities [17]. Several works which have been conducted to compute these metrics rapidly exist in the literature. The algorithm with the best asymptotic complexity to compute centrality metrics [2] is believed to be asymptotically optimal [9]. Research have focused on either approximation algorithms for computing centrality metrics [3, 6, 15] or on high performance computing techniques [12, 20]. It is common to find large social networks, and we are always in a quest for better techniques and algorithms which help us to cope with today's large graphs.

Although betweenness centrality (BC) has been proved to be successful for network analysis, computing the centrality scores of all the nodes in a network is expensive. Brandes proposed an algorithm for computing BC with $\mathcal{O}(nm)$ and $\mathcal{O}(nm+n^2 \log n)$ time complexity and $\mathcal{O}(n+m)$ space complexity for unweighted and weighted networks, respectively, where n is the number of nodes in the network and m is the number of node-node interactions in the network [2]. Brandes' algorithm is currently the best algorithm for BC computations and it is unlikely that general algorithms with better asymptotic complexity can be designed [9]. However, it is not fast enough to handle Facebook's billion or Twitter's 200 million users.

To the best of our knowledge there are two main approaches to compute BC on GPUs: the vertex-based approach which assigns a single thread to a vertex [20, 7], and the edge-based one which assigns a single thread to an edge [7]. Although the former uses comparably less memory and perform well for almost-regular graphs with less degree variance, as Jia et al. showed, the edge-based approach can improve the GPU throughput with better load balancing especially for scale-free networks [7].

In this work, we investigate a set of novel techniques to make BC computations faster with CUDA on GPUs and heterogeneous CPU/GPU architectures. Our contributions are multifold: we first reduced the memory usage of Shi and Zhang's [20] and Jia et al.'s [7] implementations by removing redundant arrays of size n^2 and m, respectively, which were being used to hold predecessors during graph traversals. We show that, by virtualizing the vertices with high

degree, we can solve the imbalance problem of the vertex-based approach and keep the memory usage lower than the edge-based approach. We further reorganize the memory accesses of the virtual vertices to obtain a better coalescing. On the structural side, we compress the graph by iteratively removing degree-1 vertices on GPUs, a technique proposed in [1, 18]. We experimentally show that combining all these techniques and using a heterogeneous CPU/GPU architecture can make the BC computations much faster for social graphs. The source code for all the techniques is available[1].

The rest of the paper is organized as follows: In Section 2, we give the notation we used in the paper and a description of Brandes' algorithm. Section 3 describes the virtualization technique we used on GPUs. The degree-1 graph reduction technique is explained in Section 4. Section 5 gives the results of the experiments, and Section 6 concludes the paper.

2. BACKGROUND

Let $G = (V, E)$ be a network modeled as a simple graph with $n = |V|$ vertices and $m = |E|$ edges where each node is represented by a vertex in V, and a node-node interaction is represented by an edge in E. Let $\Gamma(v)$ be the set of vertices which are connected to v. A vertex v is a *degree-1* vertex if and only if $|\Gamma(v)| = 1$.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *path* is a sequence of vertices such that there exists an edge between consecutive vertices. A path between two vertices s and t is denoted by $s \rightsquigarrow t$. Two vertices $u, v \in V$ are *connected* if there is a path from u to v. If all vertex pairs are connected we say that G is *connected*. If G is not connected, then it is *disconnected* and each maximal connected subgraph of G is a *connected component*, or a component, of G.

2.1 Betweenness Centrality

Given a connected graph G, let σ_{st} be the number of shortest paths from a source $s \in V$ to a target $t \in V$. Let $\sigma_{st}(v)$ be the number of such $s \rightsquigarrow t$ paths passing through a vertex $v \in V$, $v \neq s, t$. Let the *pair dependency* of v to s, t pair be the fraction $\Delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The betweenness centrality of v is defined by

$$\texttt{bc}[v] = \sum_{s \neq v \neq t \in V} \Delta_{st}(v). \tag{1}$$

Since there are $\mathcal{O}(n^2)$ pairs in V, one needs $\mathcal{O}(n^3)$ operations to compute $\texttt{bc}[v]$ for all $v \in V$ by using Equation (1). Brandes reduced this complexity and proposed an $\mathcal{O}(mn)$ algorithm for unweighted networks [2]. The algorithm is based on the accumulation of pair dependencies over target vertices. After accumulation, the dependency of v to $s \in V$ is

$$\Delta_s(v) = \sum_{t \in V} \Delta_{st}(v). \tag{2}$$

Let $\texttt{P}_s(u)$ be the set of u's predecessors on the shortest paths from s to all vertices in V. That is,

$$\texttt{P}_s(u) = \{v \in V : (v, u) \in E, \texttt{d}_s(u) = \texttt{d}_s(v) + 1\}$$

where $\texttt{d}_s(u)$ and $\texttt{d}_s(v)$ are the shortest distances from s to u and v, respectively. $\texttt{P}_s$ defines the *shortest paths graph* rooted in s. Brandes observed that the accumulated dependency values can be computed recursively:

$$\Delta_s(v) = \sum_{u : v \in \texttt{P}_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} \times (1 + \Delta_s(u)) \tag{3}$$

To compute $\Delta_s(v)$ for all $v \in V \setminus \{s\}$, Brandes' algorithm uses a two-phase approach: First, a breadth first search (BFS) is initiated from s to compute σ_{sv} and $\texttt{P}_s(v)$ for each v. Then, in a *back propagation* phase, $\Delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner by using Equation (3). In this work, we use the same algorithm but restructure the recursion to reduce the number of arithmetic operations. Let

$$\begin{aligned}
\delta_s(v) &= \frac{1 + \Delta_s(v)}{\sigma_{sv}} \\
&= \frac{1}{\sigma_{sv}} + \frac{\Delta_s(v)}{\sigma_{sv}} \\
&= \frac{1}{\sigma_{sv}} + \sum_{u : v \in \texttt{P}_s(u)} \frac{1 + \Delta_s(u)}{\sigma_{su}} \\
&= \frac{1}{\sigma_{sv}} + \sum_{u : v \in \texttt{P}_s(u)} \delta_s(u).
\end{aligned} \tag{4}$$

In Algorithm 1, we first compute $\delta_s(v)$ for each $v \in V$ and then update the BC value by using

$$\Delta_s(v) = \sigma_{sv} \times \delta_s(v) - 1 \tag{5}$$

to satisfy Equation (1) and Equation (2).

In the first phase, the algorithm computes $\sigma[v]$ for $v \in V$ which is the number of shortest paths from the source vertex s to v. In addition, the predecessors of v on these shortest paths are stored in $\texttt{P}[v]$. To compute Equation (4) in the second phase, the algorithm initiates $\delta[v]$ with $\frac{1}{\sigma[v]}$ for each $v \in V$ (line 3 of Algorithm 1). Then it adds $\delta[w]$ to $\delta[v]$ for each successor w of v (line 4 of Algorithm 1). And it increases the BC score of v by $(\delta[v] \times \sigma[v] - 1)$ (line 5 of Algorithm 1) as required by Equation (5).

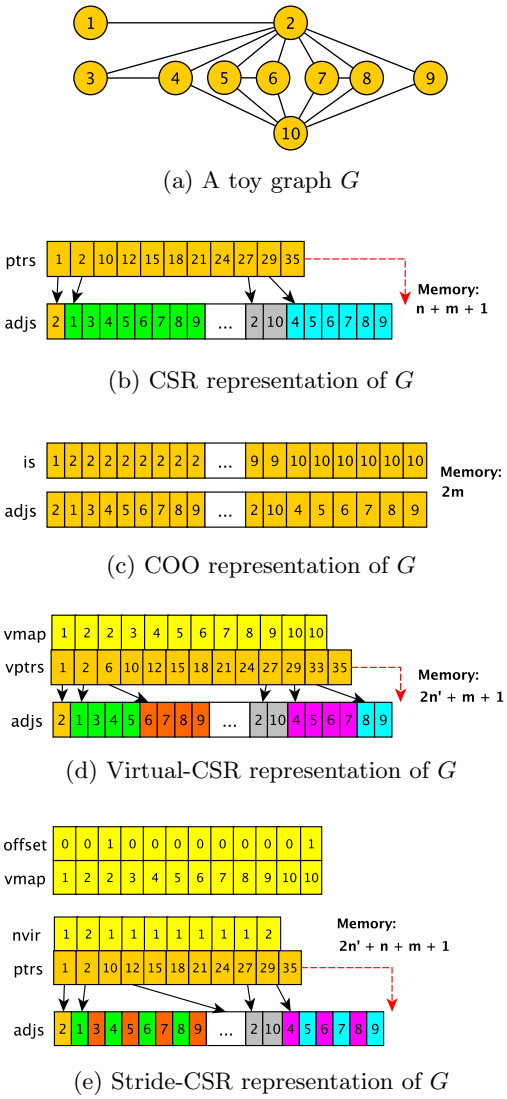
Each phase of Algorithm 1 considers all the edges at most once, taking $\mathcal{O}(m + n)$ time. The phases are repeated for each source vertex. The overall complexity is $\mathcal{O}(mn)$.

There are two parallelization options for BC: coarse- and fine-grain. In coarse-grain parallelism, the loop at line 1 of Algorithm 1 is shared among threads, i.e., the contribution of each source is computed by a single thread. With this approach, each thread must use its own memory for bc, σ, δ, P, and d to avoid read/write conflicts. But the threads work independently from each other and there is no other synchronization overhead. In fine-grain parallelism, a BFS is concurrently executed by multiple threads. Although this approach uses a single copy of each array, synchronization is not free since the conflicts need to be resolved. Furthermore, sparsity and other irregularities in the graph such as pattern and degree distribution can damage load balance and cache locality. Hence, it may be better to investigate and employ coarse-grain parallelism on devices with large cache and memory such as CPUs, whereas a fine-grain parallelization is usually the better option with restricted memory devices such as GPUs, especially when the graphs are large. However, algorithms for sparse matrices and graphs are known to be memory bound. Hence, the speedups one can achieve with GPUs is less than that of algorithms with dense and regular data.

[1] http://bmi.osu.edu/hpc/software/gpuBC

Algorithm 1: Sequential BC

```
   Data: G = (V, E)
   bc[v] ← 0, ∀v ∈ V
1  for each s ∈ V do
       S ← empty stack, Q ← empty queue
       P[v] ← empty list, σ[v] ← 0, d[v] ← −1, ∀v ∈ V
       Q.push(s), σ[s] ← 1, d[s] ← 0
       ▷Forward phase: BFS from s
       while Q is not empty do
           v ← Q.pop(), S.push(v)
           for all w ∈ Γ(v) do
               if d[w] < 0 then
                   Q.push(w)
                   d[w] ← d[v] + 1
               if d[w] = d[v] + 1 then
2                  σ[w] ← σ[w] + σ[v]
                   P[w].push(v)
       ▷Backward phase: Back propagation
3      δ[v] ← 1/σ[v], ∀v ∈ V
       while S is not empty do
           w ← S.pop()
           for v ∈ P[w] do
4              δ[v] ← δ[v] + δ[w]
       ▷Update bc values by using Equation (5)
       for v ∈ V do
           if v ≠ s then
5              bc[v] ← bc[v] + (δ[v] × σ[v] − 1)
   return bc
```



(a) A toy graph G

(b) CSR representation of G

(c) COO representation of G

(d) Virtual-CSR representation of G

(e) Stride-CSR representation of G

Figure 1: A toy graph G with 10 vertices and 17 edges (a), its CSR representation (b), its COO representation (c), its virtual-CSR representation (d), and stride-CSR representation (e). In the figures, n is the number of vertices, n' is the number of virtual vertices, and m is the number of edges. For virtualization in (c) and (d), $mdeg = 4$ is used. The memory usage of each representation is given in terms of the number of entries it has.

3. BETWEENNESS CENTRALITY ON GPU

As mentioned above, there are two existing studies on computing betweenness centrality by using GPUs. In the first one, Shi and Zhang developed a software package *gpu-fan*[2] to do biological network analysis [20]. Later, Jia et al. compared vertex- and edge-based techniques on GPUs for BC computations [7]. Both of these works employ a fine-grain parallelism: all threads work concurrently while executing a single, level-synchronized BFS. That is all the *frontier* vertices at current level ℓ must be processed before the vertices at level $\ell+1$. And for each level, the algorithms initiate a GPU kernel to visit the vertices/edges on that level. The difference between the vertex- and edge-based parallelism arises from the implementation of a forward/backward-step and the corresponding graph storage scheme. To ease the memory accesses, the former uses the compressed sparse row (CSR) format, and the latter uses the coordinate (COO) format for graph storage. Figures 1(b) and 1(c) show these storage schemes for a toy graph with 10 vertices and 17 edges as given in Figure 1(a).

3.1 Vertex-based parallelism

In vertex-based parallelism, all the edges of a single vertex are processed by a single thread. The pseudocode of the forward and backward phases of the vertex-based approach are given in Algorithm 2. Let u be a frontier vertex at level ℓ and v be one of its neighbors. There can be three cases for v: if $\mathtt{d}[v] = -1$ then v is unvisited before and will be a frontier vertex in level $\ell+1$. In this case, the kernel understands that the next frontier will not be empty and sets *cont* to **true**. It also increases $\sigma[v]$ by $\sigma[u]$, since all shortest paths from the source vertex to u will be a prefix of at least one shortest path from s to v (line 4 of Algorithm 2). This operation must be atomic since there can be other threads concurrently trying to update $\sigma[v]$. Hence, in vertex-based parallelism a single atomic operation per successor-predecessor edge is necessary. If v has been visited before, it can be either at level ℓ or $\ell-1$. In the latter case, the kernel sets u as one of the predecessors of v, i.e., $\mathtt{P}_v[u] \leftarrow 1$. To store the predecessor information, P, Shi and Zhang used an $n \times n$-bit array. Considering the size of real-world networks and graphs and the amount of memory available on modern GPUs, this is not practical even for mid-size networks. The n^2 storage is actually an overkill since a successor-predecessor relationship can be established only by an edge and there are only

[2] http://bioinfo.vanderbilt.edu/gpu-fan/

$m \ll n^2$ of them. To store the same information, Jia et al. used an array of size m. For an edge $e \in E$, indexed as in the order of CRS `adj` array, they set $\mathtt{P}[e]$ to 1 if e is a successor-predecessor edge and leave it 0, otherwise.

Let u be a vertex at level ℓ, when u is being processed in the backward-step kernel, it gathers all $\delta[v]$s from its successor vertices, i.e., all $v \in V$ such that $\mathtt{P}_v[u] = 1$. As Figure 1 shows, the vertex-based approach requires $n+m+1$ memory in total to store the graph. Here and in the rest of the paper, the memory usage of each graph representation is given in terms of the number of entries it contains.

Algorithm 2: VERTEX: vertex-based parallel BC

```
   ...
   ℓ ← 0
   ▷Forward phase
   while cont = true do
   |   cont ← false
   |   ▷Forward-step kernel
   |   for each u ∈ V in parallel do
1  |   |   if d[u] = ℓ then
2  |   |   |   for each v ∈ Γ(u) do
3  |   |   |   |   if d[v] = −1 then
   |   |   |   |   |   d[v] ← ℓ + 1, cont ← true
   |   |   |   |   else if d[v] = ℓ − 1 then P_v[u] ← 1
4  |   |   |   |   if d[v] = ℓ + 1 then σ[v] ←(atomic) σ[v] + σ[u]
   |   ℓ ← ℓ + 1
   ...
   ▷Backward phase
   while ℓ > 1 do
   |   ℓ ← ℓ − 1
   |   ▷Backward-step kernel
   |   for each u ∈ V in parallel do
   |   |   if d[u] = ℓ then
5  |   |   |   for each v ∈ Γ(u) do
6  |   |   |   |   if P_v[u] = 1 then δ[u] ← δ[u] + δ[v]
   ▷Update bc values by using Equation (5)
   ...
```

3.2 Edge-based parallelism

A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. That is there are many vertices with a degree that is lower than average, and there are some with very high degrees, yielding a very skewed degree distribution. Social networks we have today fit to this definition. And others such as collaboration networks, semantic networks, and protein-protein interaction networks also do. In a GPU, the threads in a *warp* run at the same time and must wait for each other to finish. When the variance of the degrees is high and vertex-based parallelism is used, a warp's threads most likely have an imbalanced load distribution. Edge-based parallelism is proposed to cope with this problem and as our experimental results show, it performs much better on scale-free graphs. The pseudocode of the forward and backward phases of the edge-based approach are given in Algorithm 3 in which different edges of the same vertex will be processed by different threads. The algorithm uses the COO format (as shown in Figure 1) in order to have an easy access to individual edges.

Processing a neighbor in the forward phase is similar to that of the vertex-based approach (lines 3–4 of Algorithm 2). But the number of times the line 1s executed in Algorithms 2 and 3 is different. Since there are n vertices and m edges, the number of memory accesses due to this line is more in the edge-based approach. However, for social networks, m is in the order of $\mathcal{O}(n)$, and most of these memory accesses will be coalesced since, as shown in Figure 1(c), the next value in `is` array is either the same or one more.

Although the updates in the backward-phase of the vertex-based approach are handled without using atomic instructions, in edge-based parallelism, when $\mathtt{P}_v[u] = 1$ for an edge (u, v) which is currently being processed, the update operation on $\delta[u]$ must be atomic. Because, there can be other successor-predecessor edges $(u, v') \in E$ being processed concurrently by other threads. In total, two atomic operations per successor-predecessor relationship are needed in edge-based parallelism. Hence, the edge-based approach uses both more memory and more atomic operations than the vertex-based one. But it benefits from better memory coalescing and better load distribution.

Algorithm 3: EDGE: edge-based parallel BC

```
   ...
   ℓ ← 0
   ▷Forward phase
   while cont = true do
   |   cont ← false
   |   ▷Forward-step kernel
   |   for each (u, v) ∈ E in parallel do
1  |   |   if d[u] = ℓ then
   |   |   |   ... ▷same as vertex-based forward step
   |   ℓ ← ℓ + 1
   ...
   ▷Backward phase
   while ℓ > 1 do
   |   ℓ ← ℓ − 1
   |   ▷Backward-step kernel
   |   for each (u, v) ∈ E in parallel do
   |   |   if d[u] = ℓ then
2  |   |   |   if P_v[u] = 1 then δ[u] ←(atomic) δ[u] + δ[v]
   ▷Update bc values by using Equation (5)
   ...
```

3.3 Vertex virtualization for BC

The vertex-based parallelism suffers from load balancing, and the edge-based parallelism uses more memory and more atomic operations. Here, we propose a vertex virtualization technique to alleviate both of these problems at the same time. The technique replaces the high-degree vertices with a number of virtual vertices each having at most *mdeg* (maximum degree) edges and uses

$$n' = \sum_{v \in V} \left\lceil \frac{|\Gamma(v)|}{mdeg} \right\rceil$$

virtual vertices in the modified graph storage scheme. The value of *mdeg* can be chosen by the user. In our preliminary experiments, we have experimented with various *mdeg* values (2, 4, and 8), and the results were close to each other but 4 was just slightly better, hence in our experiments we used this value.

Figure 1(d) shows the virtual-CSR representation for the toy graph in Figure 1(a) after virtualization with $mdeg = 4$. There are three arrays: `vmap` maps the virtual vertices to real vertices in V, `vptrs` is similar to the `ptrs` in CSR and each entry shows the first neighbor of the corresponding virtual vertex in `adjs` which is the same as the one in the traditional CSR.

The pseudocode of the forward and backward phases of the virtualization-based approach are given in Algorithm 4.

Algorithm 4: VIRTUAL: BC with virtual vertices

```
...
ℓ ← 0
▷Forward phase
while cont = true do
    cont ← false
    ▷Forward-step kernel
    for each virtual vertex u_vir in parallel do
        u ← vmap[u_vir]
        if d[u] = ℓ then
1           for each v ∈ Γ_vir(u_vir) do
2               if d[v] = −1 then
                    d[v] ← ℓ + 1, cont ← true
3               if d[v] = ℓ + 1 then σ[v] ←(atomic) σ[v] + σ[u]
    ℓ ← ℓ + 1
...
▷Backward phase
while ℓ > 1 do
    ℓ ← ℓ − 1
    ▷Backward-step kernel
    for each virtual vertex u_vir in parallel do
        u ← vmap[u_vir]
        if d[u] = ℓ then
            sum ← 0
4           for each v ∈ Γ(u) do
5               if d[v] = ℓ + 1 then sum ← sum + δ[v]
6           δ[u] ←(atomic) δ[u] + sum
▷Update bc values by using Equation (5)
...
```

In the forward phase, each thread processes the edges of a virtual vertex u_{vir}. The real vertex u is reached via `vmap` and $\sigma[u]$ is used to update the number of shortest paths to neighbors of u_{vir}. In the backward phase, a similar approach is used.

Notice that Algorithm 4 does not store the predecessor-successor edges. Instead it checks if $\mathtt{d}[v] \stackrel{?}{=} \mathtt{d}[u]+1$ to decide whether an edge (u, v) is a predecessor-successor edge or not. Our preliminary experiments showed no significant difference in runtime between storing the predecessor-successor information or using the distances. Therefore, we chose not to store it in any of our GPU implementations since we can save a significant amount of memory.

There are four main advantages of virtual-CSR over the other representatives:

1. When *mdeg* is small, e.g., 4, the load imbalance among the threads in a warp will not be as troublesome as the one in traditional CSR.

2. Only the threads which are processing the frontier vertices are active during a forward/backward step in the vertex-based approach. Hence, the idle threads need to wait others. In virtual-CSR, the virtual vertices which represent the same vertex $v \in V$ are labeled consecutively and probably, they will be in the same warp. Hence, when v is in the frontier, all its virtual vertices in the virtual-CSR will be frontier too. This probably will increase the average ratio of the number of active vertices in a warp.

3. As shown in Algorithm 3, the updates on δ in the backward phase must be atomic when the edge-based parallelism is used. The number of such atomic operations is equal to the number of predecessor-successor edges in E. Usually, a high fraction of the edges are such edges in practice. With virtualization, the number of atomic operations in the backward phase is reduced to n' (line 6 of Algorithm 4) since the all the updates are computed and stored in a local variable *sum* (line 5 of Algorithm 4).

4. In total, the virtual-CSR storage uses $2n'+m+1$ memory. We can assume that n' is in the order of $m/mdeg$. Hence, compared with the edge-based approach, when $mdeg = 4$, this saves an amount of memory of approximately $m/2$.

3.3.1 Stride-CSR representation

As we will experimentally show, the virtual-CSR representation is effective and efficient. But it can be further improved by reorganizing the memory accesses to `adjs`. As mentioned above, the virtual vertices are consecutively labeled and the threads processing them will probably be in the same warp. Each such thread accesses a continuous part of `adjs` of size at most *mdeg*. It means that the threads in a warp access `adjs` in a regularly spaced fashion over a range of $warpsize \times mdeg$. Such accesses are unlikely to be coalesced.

The stride-CSR representation (shown in Figure 1(e)) allows better coalescing. In addition to original CSR arrays `ptrs` and `adjs`, it needs three additional arrays: `vmap` is the same mapping array used in virtual-CSR, `offset` shows the virtual vertex number within the real vertex, and `nvir` shows the number of virtual vertices for each vertex in V. By using this additional information we distribute the edges of v to its virtual vertices in a round robin fashion. Hence, the memory accesses by consecutive threads, which process these virtual vertices, will also be consecutive and better coalesced. Although it uses more memory, experimental results show that the memory access scheme of stride-CSR is superior to that of virtual-CSR for many graphs. From now on, we will call the version of the code that uses stride-CSR as STRIDE.

4. COMPRESSING THE GRAPH FOR BC

Brandes' algorithm relies on the shortest path graph rooted on a given source of the graph. Two different sources can expose almost the same shortest path graph. And in some cases, it is possible to exploit this and other structures by considering both of them simultaneously [1, 18].

In particular, we are interested in removing vertices with exactly one neighbor, i.e., degree-1 vertices. Let $u \in V$ be such a vertex with a neighbor $v \in V$. If u lies on a shortest path in G, it must be one of the endpoints. Hence, BC of u is 0. However, the vertex can not be just removed since $\delta_{su}(w)$ is not necessarily equal to 0 for all $w \in V$. All of the shortest paths that go to u must be through v; indicating that $\delta_{su}(w) = \delta_{sv}(w)$, for all $w \neq v \in V$. Notice that the expression is not correct for $w = v$ since $\delta_{sv}(v) = 0$ and $\delta_{su}(v) = 1$: the shortest path to v does not go through v (but the paths to u go through v). Because the graph is undirected, a similar relation holds between $\delta_{us}(w)$ and $\delta_{vs}(w)$.

We introduce the `reach` array, which indicates that vertex w represents `reach`$[w]$ vertices (including itself). The difference in the BC computation only changes two lines of Algorithm 1. The initialization of $\delta[v]$ values (line 3 of Algorithm 1) should be multiplied by `reach`$[v]$ to account for

shortest paths going to vertices represented by v. The increment to $\texttt{bc}[w]$ (line 5 of Algorithm 1) should be multiplied by $\texttt{reach}[s]$ to account for shortest paths going from vertices represented by s.

All the values of $\texttt{reach}[w]$ are initialized to 1. When a degree-1 vertex u connected to vertex v is detected, $\texttt{reach}[v]$ is incremented by $\texttt{reach}[u]$ and u is removed from the graph. This operation can be applied iteratively in order to remove all the "terminal trees" in the graph. When u is removed from the graph, the BC values of u and v need to be updated to take into account paths that come from vertices represented by u to v (or through v) according to the following formulas:

$$\texttt{bc}[u] += (\texttt{reach}[u] - 1) * (|V| - \texttt{reach}[u]) \quad (6)$$

$$\texttt{bc}[v] += (|V| - \texttt{reach}[u] - 1) * \texttt{reach}[u] \quad (7)$$

All the proofs and details can be found in [18].

4.1 Implementation on GPU

The modifications to the BC kernels are minor and are omitted. We only discuss in this section how the preprocessing in implemented in GPU within CUDA.

```
__global__ void rm_deg1(int* xadj, int* adj, int* tadj, int n,
                        float* bc, int* reach, bool *cont, int* deg)
{
    int u = blockIdx.x * blockDim.x + threadIdx.x;

    if (u < n) {
        if (deg[u] == 1) {
            int vminr = n - reach[u];
            bc[u] += (reach[u] - 1) * vminr;
            *cont = true;
            deg[u] = 0;

            int end = xadj[u + 1];
            for (int p = xadj[u]; p < end; p++) {
                int v = adj[p];
                if (v != -1) {
                    adj[p] = -1;
                    adj[tadj[p]] = -1;

                    atomicAdd(bc + v, reach[u] * (vminr - 1));
                    atomicAdd(reach + v, reach[u]);
                    atomicAdd(deg + v, -1);
                    break;
                }
            }
        }
    }
}
```

Figure 2: CUDA kernel for degree-1 vertices identification and removal.

The identification and removal of degree-1 vertices is performed by calling the CUDA kernel in Figure 2. Since a call to the kernel might uncover new degree-1 vertices, the kernel is called iteratively until no more vertices are removed (i.e., until $*cont$ is **false**). One thread per vertex is used and all the threads that do not operate on a vertex u whose degree is 1 terminates immediately. This operation is performed in constant time by keeping track of the degrees of the vertices explicitly. Then, the thread adjusts the bc value of the degree one vertex and sets its degree to 0. The thread then searches for the vertex v, it is attached to. The edge that connect u to v is set to -1 to deactivate it. Notice that in the CSR representation, that each undirected edge appears as two directed edge, one in the neighbors of u and one in the neighbors of v. Finally, 3 atomic operations are performed to update $\texttt{bc}[v]$, $\texttt{reach}[v]$ and the degree of v. Notice that because the degree of v is updated last, no race condition is possible due to the non atomic access to $\texttt{bc}[u]$.

Once there are no more degree-1 vertices in the graph, the adjacency list of the graph is compacted on the CPU.

5. EXPERIMENTS

The experimental studies were carried out on the compute nodes of our in-house cluster. Each node is equipped with two Intel Xeon E5520 CPU clocked at 2.27Ghz and 48GB of memory, split across the two NUMA domains. Each CPU is a quad-core and HyperThreading is enabled. Each core has its own 32kB L1 cache and 256kB L2 cache. The 4 cores on a CPU share a 8MB L3 cache. Each node is also equipped with an NVIDIA Tesla C2050 with usable 2.6GB of global memory. C2050 is equipped with 14 multiprocessors which contains 32 CUDA cores each, for a total of 448 CUDA cores. The CUDA cores are clocked at 1.15GHz and the memory is clocked at 1.5GHz. ECC is enabled.

On the software side, the node runs 64-bit CentOS with Linux 2.6.32-71.el6. All the codes are compiled with GCC 4.4.4 and CUDA 4.2.9 with the -O2 optimization flag and -arch sm_20. We have carefully implemented all the algorithms using C. To have a base-line comparison, we tried to run *gpu-fan* [20] on our test set. Due to the high memory usage of this work, we could run it on the smallest graph we have in our experiment set, and its execution time, was even much slower than our carefully optimized CPU code, hence we did not include the results here.

To test the algorithms, we extracted graphs from the SNAP dataset[3]. We selected 8 of the largest social network graphs that will fit on our GPUs. Directed graphs were made undirected and the largest connected component is extracted. The list of graphs and the properties of the largest components that are used in our experiments can be found in Table 1. Table also lists the properties of the component after degree-1 reduction. The distribution of the degrees can be seen in Figure 3.

All the results presented in this section are total application time from the moment where the graph is fully loaded in the main memory of the machine to the moment where the final BC values are available on the main memory of the node. In particular, the time excludes reading the graph from the hard drive; but it includes all the transformations performed on the graph (such as ordering, reduction and format conversion) and it includes all the communications between the host and the device. Since the computations can be extremely long (months), we measured the time for a given number of sources/BFSs and extrapolated linearly to the runtime for all the graph, paying attention to take properly into account the constant one-time overheads of the application. Depending on the speed of the configuration, we run one thousand, ten thousand, or hundred thousand sources which provide a runtime large enough to smoothen any system fluctuations. The runtime of single source proved to be very stable, allowing us to make a meaningful extrapolation. Table 2 displays the sequential execution times of BC computations, as well as various parallel configurations. These results, and more, will be discussed in more detailed in following subsections.

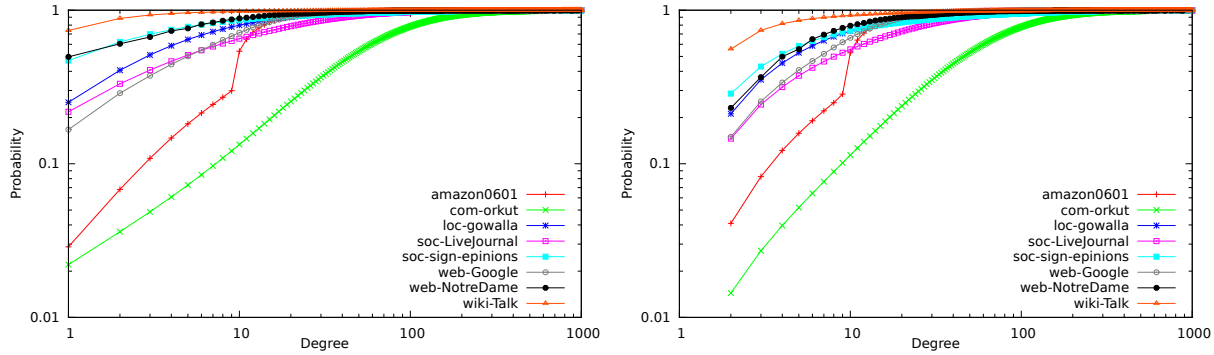
[3] http://snap.stanford.edu/data/index.html

Graph	Original				Reduced			
	$\|V\|$	$\|E\|$	avg $\|\Gamma(v)\|$	max $\|\Gamma(v)\|$	$\|V\|$	$\|E\|$	avg $\|\Gamma(v)\|$	max $\|\Gamma(v)\|$
amazon0601	403,364	4,886,622	12.1	2,752	390,915	4,861,724	12.4	2,750
com-orkut	3,072,441	234,370,166	76.2	33,313	3,004,647	234,234,578	77.9	33,275
loc-gowalla	196,591	1,900,654	9.6	14,730	142,670	1,792,812	12.5	14,730
soc-LiveJournal	4,843,953	85,691,368	17.6	20,333	3,740,572	83,484,606	22.3	20,329
soc-sign-epinions	119,130	1,408,534	11.8	3,558	59,288	1,288,850	21.7	3,461
web-Google	855,802	8,582,704	10.0	6,332	703,942	8,278,984	11.7	6,292
web-NotreDame	325,729	2,180,216	6.6	10,721	159,683	1,848,124	11.5	10,721
wiki-Talk	2,388,953	9,313,364	3.8	100,029	622,986	5,781,430	9.2	46,241

Table 1: Properties of the largest component of the graph in our dataset before and after degree-1 reduction.

Graph	CPU 1 thread	CPU 8 threads deg1 + ord	GPU stride deg1 + ord	Heterogeneous stride deg1 + ord
amazon0601	46,297	5,883	4,535	2,321
com-orkut	10,862,615	2,204,789	2,016,763	1,077,391
loc-gowalla	6,278	533	598	290
soc-LiveJournal	10,614,646	1,197,589	858,735	515,524
soc-sign-epinions	2,269	118	169	78
web-Google	157,920	9,652	8,912	4,678
web-NotreDame	6,847	304	824	234
wiki-Talk	465,755	13,426	7,167	4,456

Table 2: Total execution time of betweenness centrality on various configurations (in seconds).



(a) Original (b) Reduced

Figure 3: Cumulative density functions of the vertex degrees of the test graphs.

5.1 GPU implementations

We first present the comparison of the four GPU implementations on the original graphs in Figure 4. The results are presented in terms of speedups compared to a sequential CPU implementation (see Table 2). On these graphs, there is a clear ranking of the four implementations, vertex-based parallelism (VERTEX) is typically slower than edge-based parallelism (EDGE). And both of these existing methods were outperformed by the two methods proposed in this paper, that are based on virtual vertices (VIRTUAL) and strided memory accesses (STRIDE).

VERTEX can be more than 5 times slower than EDGE (e.g., on loc-gowalla and wiki-Talk). STRIDE is never slower than VIRTUAL, (it is just about 3% faster on amazon0601, web-Google, and web-NotreDame) and it can be up to 17% faster (on soc-livejournal) thanks to better memory coalescing. On average, STRIDE is 1.6 times faster than EDGE and 3.7 times faster than VERTEX.

From Figure 4 it might look like the performance of the GPU implementations are particularly bad on web-NotreDame. Remember that these are speedups relative to a sequential CPU. Figure 5 displays the absolute performance expressed in number of edges processed per second (computed as $|V||E|/runtime$) for both CPU and GPU (STRIDE) codes. The performance of the GPU STRIDE on web-NotreDame is similar to the one achieved on other graphs. However, the sequential CPU implementation is significantly more efficient on that particular graph, most likely due to structure of the graph and ordering of the vertices that yield more cache-friendly execution, and hence the lower speedup values.

5.2 Graph modifications

We now consider the impact of the reduction of the graph by removing degree-1 vertices. We also consider the impact of ordering the graph. We order the graph with respect to a queue ordering in a single BFS computed from an arbitrary vertex. This is similar to the Reverse Cuthill-McKee ordering which is popular in the context of sparse linear algebra [5]. Intuitively, such an ordering put vertices at the same level together. Since the graph is traversed in a level-by-level fashion (but not rooted in the same source), the ordering should improve the percentage of active threads in

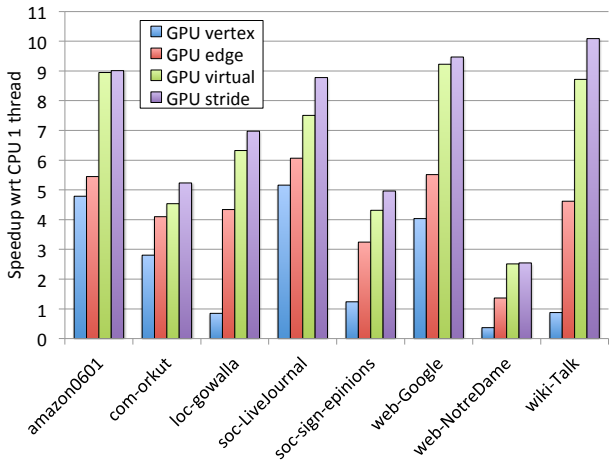


Figure 4: Comparison of GPU implementations.

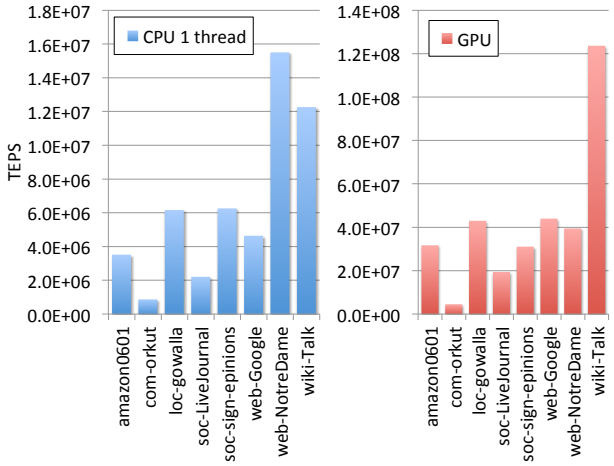


Figure 5: Absolute performance for sequential CPU and GPU stride expressed in Traversed Edge Per Second (TEPS)

a warp, the number of completely empty blocks, and minimize the overhead due to thread divergence.

Figure 6(a) shows the impact for the sequential CPU case. In the CPU, graph reduction has almost no impact on com-orkut but it brings a 7-fold improvement on wiki-Talk. On average, graph reduction brings a 2-fold improvement. Graph ordering brings barely any improvement on two graphs, but it brings a 53% improvement on web-google. When graph reduction and ordering combined, total graph modifications bring up to 7.6-fold improvement, with an average of 2.21.

Figure 6(b) shows the impact of the graph manipulation for GPU STRIDE implementation. The behavior is similar to the one observed on the CPU. The most notable difference is that ordering harms performance on soc-sign-epinions and wiki-Talk. Though, it brought a 35% improvement on web-Google.

Note that there are two sources of improvement that spurs from graph reduction. First, since there are less vertices, there are less source to execute. Second, the graph is smaller which makes each source faster.

5.3 Heterogeneous execution

In the last set of experiments, we evaluated performance of using CPU alone, GPU alone, and using CPU and GPU together for BC computation. Figure 7 shows the performance obtained by using only CPUs (8 CPU threads), using only GPU (proposed two methods VIRTUAL and STRIDE presented) and using both the CPUs and the GPU at the same time (labeled as "Heterogeneous"). Notice that in the later (heterogeneous) case, we utilize only 7 threads on the CPU to dedicate one core to drive the GPU.

The source based parallelism used on the CPU shows an average parallel speedup of 6 which indicates that the parallel CPU implementation, even though not linear, is fairly efficient. (Figure 7 shows an average speedup of 13, but there is a factor of 2.2 which comes from graph modifications and not parallelism.)

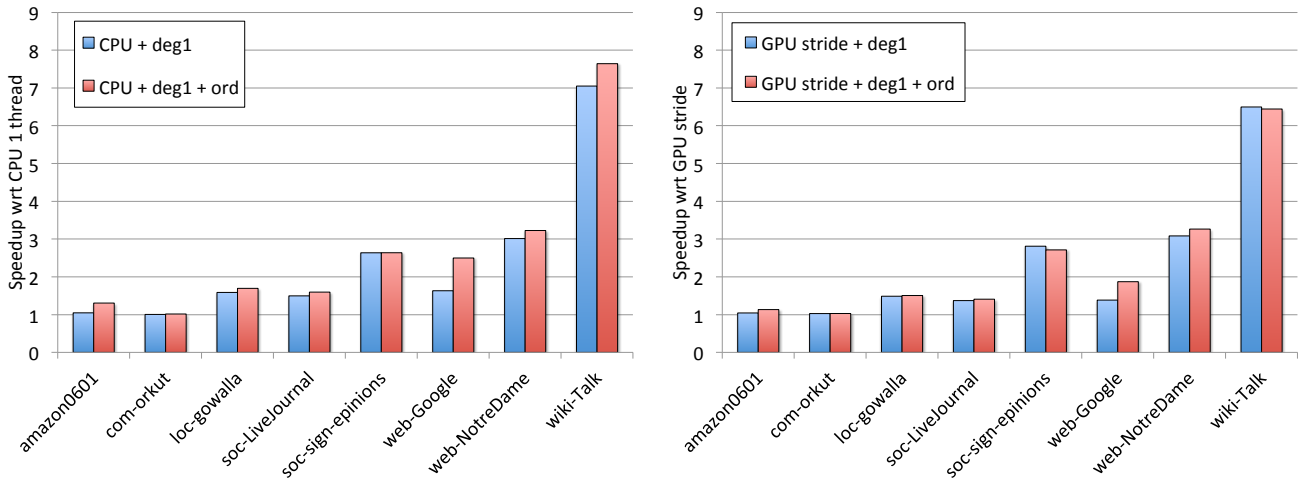
The GPU STRIDE implementation reaches higher performance than the parallel CPU implementation in 5 graphs (amazon0601, com-orkut, soc-LiveJournal, web-Google, and wiki-Talk), while the CPU implementation obtains higher performance on 3 graphs (web-Google, soc-sign-epinions and loc-gowalla). If one computes geometric mean, on average the parallel CPU implementation and the GPU implementation reach the same performance (less than 1% difference in the average). This indicates that the correct choice between CPU and GPU for betweenness centrality is strongly input dependent which makes a heterogeneous collaboration between CPU and GPU important.

Using both the CPU and the GPU allows to reach the highest performance in all the graphs of our dataset. It improves the best mono-device performance by a factor a 1.29 on web-NotreDame where the performance of the CPU and GPU are the most different and by a factor of 1.95 on amazon0601 where the performance of both the CPU and GPU are the most similar.

6. CONCLUSION AND FUTURE WORK

In this work, we investigated a set of techniques to speed up the betweenness centrality computation on GPUs and CPU/GPU heterogeneous architectures. Our techniques include leveraging the topological properties of graph, i.e., compressing by removing degree-1 vertices, as well as utilizing the architectures efficiently. We provided four different GPU algorithms and compared them experimentally. Combining all the techniques yield a 104 speedup on a large social network. Our techniques in GPU algorithms can be applied to shortest-path algorithms, and compression techniques we provided can be used to speed-up graph algorithms with similar objectives with betweenness centrality.

The efficiency of our GPU implementation depends on the diameters of the graphs. In the worst case, the diameter can be n and the total work will be quadratic on the number of vertices. Social networks, in general, obey the *smallworld phenomenon* and their diameters are small. As a future work, we plan to investigate faster betweenness centrality computation techniques on graphs with large diameters by existing [11, 14] and novel techniques. Also, we plan to incorporate further graph compression techniques [18] to be used in heterogeneous architectures. Apart from that, we are planning to make more detailed analysis on proposed GPU algorithms on social networks with different characteristics, like diameter, density and degree distribution.



(a) CPU (b) GPU

Figure 6: Impact of degree-1 reduction and graph ordering

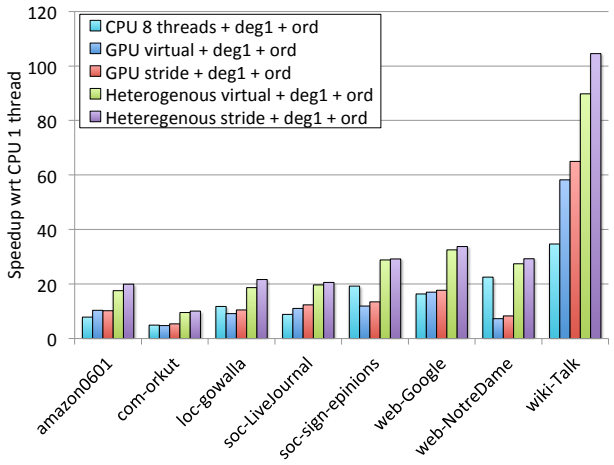


Figure 7: Comparison of CPU only, GPU only and heterogeneous execution.

Acknowledgement

This work was partially supported by the NSF grants CNS-0643969, OCI-0904809 OCI-0904802, and OCI-1246001.

7. REFERENCES

[1] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast exact computation of betweenness centrality in social networks. In *Proceedings of International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2012.

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

[3] S. Y. Chan, I. X. Y. Leung, and P. Liò. Fast centrality approximation in modular networks. In *Proceeding of the ACM First International Workshop on Complex Networks Meet Information and Knowledge Management (CIKM-CNIKM)*, pages 31–38, 2009.

[4] Ö. Şimşek and A. G. Barto. Skill characterization based on betweenness. In *Proceedings of Neural Information Processing Systems (NIPS)*, 2008.

[5] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM national conference*, 1969.

[6] D. Eppstein and J. Wang. Fast approximation of centrality. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 228–229, 2001.

[7] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. Edge vs. node parallelism for graph centrality metrics. In *GPU Computing Gems: Jade Edition*. Morgan Kaufmann, 2011.

[8] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7, 2010.

[9] S. Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.

[10] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.

[11] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[12] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[13] E. L. Merrer and G. Trédan. Centralities: Capturing the fuzzy notion of importance in social graphs. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS)*, 2009.

[14] D. Merrill, M. Garland, and A. Grimshaw. Scalable

gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[15] K. Okamoto, W. Chen, and X.-Y. Li. Ranking of closeness centrality for large-scale social networks. In *Proceedings of the 2nd annual international workshop on Frontiers in Algorithmics (FAW)*, pages 186–195, 2008.

[16] M. C. Pham and R. Klamma. The structure of the computer science knowledge network. In *Proceedings of International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2010.

[17] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and M. R. Street centrality and densities of retail and services in Bologna, Italy. *Environment and Planning B: Planning and Design*, 36(3):450–465, 2009.

[18] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM Data Mining Conference (SDM)*, 2013.

[19] X. Shi, J. Leskovec, and D. A. McFarland. Citing for high impact. In *Proceedings of the Joint Conference on Digital Library (JCDL)*, 2010.

[20] Z. Shi and B. Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.