Faster Centrality Computations on GPUs

Ümit V. Çatalyürek
Professor
Department of Biomedical Informatics
Department of Electrical & Computer Engineering
Department of Computer Science & Engineering (courtesy)
The Ohio State University
Massive networks are everywhere.

Facebook has a billion users and a trillion connections
Twitter has more than 200 million users

Who is more important in a network? Who controls the flow between nodes?

Centrality metrics answer these questions
Betweenness Centrality (BC) is an intriguing metric

BC is expensive to compute & hard to parallelize on GPUs
Centrality Metrics

- Let \( G=(V, E) \) be a graph with vertex set \( V \) and edge set \( E \)

- Degree centrality: 
  
  \[
  dc(v) = |\{\{u, v\} \in E : u \in V\}|
  \]

- Closeness centrality: 
  
  \[
  cc(v) = \sum_{u \in V \setminus \{v\}} \frac{1}{d(u, v)}
  \]

- Betweenness centrality: 
  
  \[
  bc(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}
  \]
Betweenness Centrality (BC)

• Betweenness metric models the centrality better
• We need to use all shortest paths to compute BC scores
• The current best algorithm is (by Brandes)
  • $O(|V|x|E|)$ for unweighted graphs
  • $O(|V|x|E| + |V|^2 x \log|E|)$ for weighted graphs
• Very costly considering today’s large-scale networks
  • Faster solutions are essential
    • Approximations and/or high performance computing
**Betweenness Centrality (BC)**

- Brandes’ algorithm applies two phases for each vertex $s$
  - Phase 1: simple BFS with shortest path counting
  - Phase 2: computing partial BC scores with counted paths

**Phase 1: BFS from $s$**

```plaintext
def Phase 1: BFS from $s$
while $Q$ is not empty do
    $v$ ← $Q$.pop()
    $S$.push($v$)
    for all $w \in \Gamma(v)$ do
        if $d[w] < 0$ then
            $Q$.push($w$)
            $d[w] \leftarrow d[v] + 1$
        if $d[w] = d[v] + 1$ then
            $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
            $P[w]$.push($v$)
```

**Phase 2: Back propagation**

```plaintext
def Phase 2: Back propagation
$\delta[v] \leftarrow 0, \forall v \in V$
while $S$ is not empty do
    $w$ ← $S$.pop()
    for $v \in P[w]$ do
        $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
        if $w \neq s$ then
            $bc[w] \leftarrow bc[w] + \delta[w]$
```

**GTC 2013**
Betweenness Centrality on GPUs

- Two choices; coarse-grain or fine-grain parallelism
- Coarse-grain is source-based parallelism: each source vertex is assigned to a thread
  - No synchronization problem; computations are independent
  - Separate memory for each thread (|V| times more memory!)
  - NOT feasible for large graphs and GPU
- Fine-grain parallelism is applied to a single BFS
  - Parallel execution of BFS levels
  - Synchronization inside each level: usage of atomics brings overhead
  - Thread-divergence issues due to the skewed degree distribution of real-world networks

Faster Centrality Computations on GPUs
Existing fine-grain solutions

- **Vertex-based parallelism**
  - Assigns (the edges of) each vertex to a single thread
  - Good: less number of threads
  - Bad: thread divergence problem, especially when the degree distribution is skewed (common in real-world networks)

- **Edge-based parallelism**
  - Assigns each edge to a thread
  - Good: less thread divergence
  - Bad: more work, more atomic operations
Virtual vertices

• We propose using virtual vertices
  • Use multiple virtual vertices for a vertex with high degree
  • Hybrid edge/vertex parallelism.
Graph Storage with Virtualization

- \( n \): number of vertices
- \( m \): number of edges
- \( n' \): number of virtual vertices
- maximum degree of a virtual vertex is selected as 4
- \textbf{vmap} array maps the virtual vertices to original vertices

(a) A toy graph \( G \)

(b) CSR representation of \( G \)

(c) COO representation of \( G \)

(d) Virtual-CSR representation of \( G \)
Parallel BC with Virtualization: 1st phase

\[
\ell \leftarrow 0
\]

▷ Forward phase

while \( \text{cont} = \text{true} \) do

\( \text{cont} \leftarrow \text{false} \)

▷ Forward-step kernel

for each virtual vertex \( u_{\text{vir}} \) in parallel do

\( u \leftarrow \text{vmap}[u_{\text{vir}}] \)

if \( d[u] = \ell \) then

1. for each \( v \in \Gamma_{\text{vir}}(u_{\text{vir}}) \) do

2. if \( d[v] = -1 \) then

\hspace{1cm} \( d[v] \leftarrow \ell + 1, \text{cont} \leftarrow \text{true} \)

3. if \( d[v] = \ell + 1 \) then \( \sigma[v] \leftarrow \sigma[v] + \sigma[u] \)

\( \ell \leftarrow \ell + 1 \)
Parallel BC with Virtualization: 1st phase

\[
\ell \leftarrow 0
\]

▷ Forward phase
while \( \text{cont} = \text{true} \) do
  \( \text{cont} \leftarrow \text{false} \)
  ▷ Forward-step kernel
  for each virtual vertex \( u_{\text{vir}} \) in parallel do
    \( u \leftarrow \text{vmap}[u_{\text{vir}}] \)
    if \( d[u] = \ell \) then
      for each \( v \in \Gamma_{\text{vir}}(u_{\text{vir}}) \) do
        if \( d[v] = -1 \) then
          \( d[v] \leftarrow \ell + 1, \text{cont} \leftarrow \text{true} \)
        \( d[v] \leftarrow \ell + 1 \) then
          \( \sigma[v]^{\text{atomic}} \leftarrow \sigma[v] + \sigma[u] \)
    \( \ell \leftarrow \ell + 1 \)
Parallel BC with Virtualization: 1st phase

\[
\ell \leftarrow 0 \\
\text{Forward phase} \\
\text{while } \text{cont} = \text{true} \text{ do} \\
\quad \text{cont} \leftarrow \text{false} \\
\quad \text{Forward-step kernel} \\
\quad \text{for each virtual vertex } u_{\text{vir}} \text{ in parallel do} \\
\quad \quad u \leftarrow \text{vmap}[u_{\text{vir}]} \\
\quad \quad \text{if } d[u] = \ell \text{ then} \\
\quad \quad \quad \text{for each } v \in \Gamma_{\text{vir}}(u_{\text{vir}}) \text{ do} \\
\quad \quad \quad \quad \text{if } d[v] = -1 \text{ then} \\
\quad \quad \quad \quad \quad d[v] \leftarrow \ell + 1, \text{ cont } \leftarrow \text{true} \\
\quad \quad \quad \quad \text{if } d[v] = \ell + 1 \text{ then } \sigma[v]^{\text{atomic}} \leftarrow \sigma[v] + \sigma[u] \\
\quad \ell \leftarrow \ell + 1
\]

original vertex ID

neighbors of virtual vertex are visited
• NO predecessor array, NO queue, only level information
• the avg ratio of active threads in a warp is higher
• Less thread divergence

\[ \text{if } d[v] = \ell + 1 \text{ then } \sigma[v] \leftarrow \sigma[v] + \sigma[u] \]

neighbors of virtual vertex are visited.
Parallel BC with Virtualization: 2\textsuperscript{nd} phase

\begin{algorithmic}
\Procedure{Backward phase}{\textbf{while }\ell > 1 \textbf{ do}}
\State $\ell \leftarrow \ell - 1$
\EndProcedure
\Procedure{Backward-step kernel}{\textbf{for each virtual vertex }$u_{\text{vir}}$ \textbf{ in parallel do}}
\State $u \leftarrow \text{vmap}[u_{\text{vir}}]$
\If{$d[u] = \ell$}
\State $\text{sum} \leftarrow 0$
\EndIf
\For{each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$}
\If{$d[v] = \ell + 1$}
\State $\text{sum} \leftarrow \text{sum} + \delta[v]$
\EndIf
\EndFor
\State $\delta[u] \leftarrow \delta[u] + \text{sum}$
\EndProcedure
\EndAlgorithmic
Parallel BC with Virtualization: 2\textsuperscript{nd} phase

\begin{algorithm}
\DontPrintSemicolon
\setstretch{1.2}
\SetAlgoLined
\SetKwComment{kwComment}{$\triangleright$ }{}
\caption{Backward phase}
\KwWhile{$\ell > 1$} {
    $\ell \leftarrow \ell - 1$
    \KwComment{Backward-step kernel}
    \For{each virtual vertex $u_{\text{vir}}$ in parallel} {
        $u \leftarrow \text{vmap}[u_{\text{vir}}]$
        \If{d[$u$] = $\ell$} {
            $\text{sum} \leftarrow 0$
            \For{each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$} {
                \If{d[$v$] = $\ell + 1$} {
                    $\text{sum} \leftarrow \text{sum} + \delta[v]$
                    $\delta[u] \leftarrow \delta[u] + \text{sum}$
                }
            }
        }
    }
    \For{each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$} {
        \If{d[$v$] = $\ell + 1$} {
            $\delta[v] \leftarrow \delta[v] + \text{sum}$
        }
    }
}
\end{algorithm}

original vertex ID
Parallel BC with Virtualization: 2\textsuperscript{nd} phase

\begin{algorithm}
\begin{algorithmic}
\Procedure{Backward phase}{\algorithmicdo}
\State $\ell \leftarrow \ell - 1$
\EndProcedure
\State \textbf{Backward-step kernel}
\For{each virtual vertex $u_{vir}$ in parallel}
\State $u \leftarrow vmap[u_{vir}]$
\If {$d[u] = \ell$}
\State $\text{sum} \leftarrow 0$
\For{each $v \in \Gamma_{vir}(u_{vir})$}
\If {$d[v] = \ell + 1$}
\State $\text{sum} \leftarrow \text{sum} + \delta[v]$
\EndIf
\EndFor
\State $\delta[u] \leftarrow \delta[u] + \text{sum}$
\EndIf
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

neighbors of virtual vertex are visited
Parallel BC with Virtualization: 2^{nd} phase

\[\text{Backward phase} \]
\[\text{while } \ell > 1 \text{ do} \]
\[\quad \ell \leftarrow \ell - 1 \]
\[\quad \text{Backward-step kernel} \]
\[\quad \text{for each virtual vertex } u_{vir} \text{ in parallel do} \]
\[\quad \quad u \leftarrow \text{vmap}[u_{vir}] \]
\[\quad \quad \text{if } d[u] = \ell \text{ then} \]
\[\quad \quad \quad \text{sum} \leftarrow 0 \]
\[\quad \quad \quad \text{for each } v \in \Gamma_{vir}(u_{vir}) \text{ do} \]
\[\quad \quad \quad \quad \text{if } d[v] = \ell + 1 \text{ then} \quad \text{sum} \leftarrow \text{sum} + \delta[v] \]
\[\quad \quad \quad \delta[u]^{\text{atomic}} \leftarrow \delta[u] + \text{sum} \]

for each \( v \in \Gamma_{vir}(u_{vir}) \) do

original vertex ID

eighbors of virtual vertex are visited

“sum” is used to reduce number of atomic operations
Parallel BC with Virtualization: 2\textsuperscript{nd} phase

\begin{algorithm}
\SetKwRepeat{Do}{do}{while}
\KwWhile{$\ell > 1$} {
\Do{$\ell \leftarrow \ell - 1$}{}
\Do{\textbf{Backward-step kernel}}{
\For{each virtual vertex $u_{\text{vir}}$ in parallel} {
$u \leftarrow \text{vmap}[u_{\text{vir}}]$
\If{$d[u] = \ell$} {
$\text{sum} \leftarrow 0$
\For{each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$} {
\If{$d[v] = \ell + 1$} {$\text{sum} \leftarrow \text{sum} + \delta[v]$}
$\delta[u] \leftarrow \delta[u] + \text{sum}$
}
}}}
\For{each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$} {} 
}\end{algorithm}

- Backward phase
- While $\ell > 1$
  - $\ell \leftarrow \ell - 1$
  - Backward-step kernel
    - For each virtual vertex $u_{\text{vir}}$ in parallel do
      - $u \leftarrow \text{vmap}[u_{\text{vir}}]$
      - If $d[u] = \ell$ then
        - $\text{sum} \leftarrow 0$
        - For each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$ do
          - If $d[v] = \ell + 1$ then $\text{sum} \leftarrow \text{sum} + \delta[v]$
          - $\delta[u] \leftarrow \delta[u] + \text{sum}$
    - For each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$ do

- "sum" is used to reduce number of atomic operations
- One atomic operation per virtual vertex
- Original vertex ID
- Neighbors of virtual vertex are visited

There are four main advantages of virtual-CSR over the traditional CSR representation.

1. As we will experimentally show, the virtual-CSR representation is efficient.
2. For each vertex in the traditional CSR, there is an additional virtual vertex.
3. The number of such atomic operations in the backward phase is reduced to $O(|V|)$.
4. The virtual-CSR will be frontier too. This probably results in a coalesced memory accesses by consecutive threads, which process the same virtual vertex.

There are some limitations, though. For example, we cannot store the predecessor-successor edge or the shortest path to a virtual vertex.

In the virtual-CSR, the memory accesses by consecutive threads, which process the same virtual vertex, will probably be in warp access.
Virtualization with Strided Memory Access

- **n**: number of vertices
- **m**: number of edges
- **n’**: number of virtual vertices
- maximum degree of a virtual vertex is selected as 4
- **vmap** array maps the virtual vertices to original vertices
- **adjs** and **ptrs** are changed to provide coalescing memory access

### Algorithm 1

```
for each v
    bc[v] = 0

while Q is not empty
    s = Q.pop()
    BC computations
    Q = Q + v
    while Q is not empty
        d[s] = d[s] + 1
        for each w
            if d[w] == d[s] + 1 then
                Q.push(w)

for each v
    bc[v] = bc[v] - bc[u]
```

### Figures

- **(b) CSR representation of G**
- **(c) COO representation of G**
- **(d) Virtual-CSR representation of G**
- **(e) Stride-CSR representation of G**
Degree-1 Vertex Removal and Ordering

- Two other improvements for faster BC computation
- Parallel (GPU-based) removal of degree-1 vertices in a recursive manner without violating BC scores [Sariyuce et al., SDM‘13]
  - Jack and Martin are degree-1
  - BC of those vertices is 0
  - BC of the neighbor is adjusted

- Apply BFS ordering to improve cache locality
Using hybrid parallelism (GPU+CPU) for BC computations
  + Degree-1 removal techniques and graph ordering

Coarse-grain (source-level) parallelism is used for CPUs
  + Extra memory usage is not a big problem, since number of threads is quite less than GPU’s
    - In our case, it is 8

Fine-grain parallelism is used for GPUs
  + Virtualization and strided memory access
Experiments

- Two quad-core Intel Xeon E5520 CPUs (Nehalem)
  - clocked at 2.27Ghz and have 48GB of memory
  - 32kB L1, 256kB L2, 8MB L3 cache (L3 is shared by 4-core)
- Equipped with an NVIDIA Tesla C2050 (Fermi)
  - 2.6GB of global memory
  - 14 multiprocessors each have 32 CUDA cores (448 in total)
  - CUDA cores are clocked at 1.15GHz and the memory is clocked at 1.5GHz. ECC is on.
• We used 8 social network graphs from SNAP

<table>
<thead>
<tr>
<th>Graph</th>
<th>Original</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>amazon0601</td>
<td>403,364</td>
<td>4,886,622</td>
</tr>
<tr>
<td>com-orkut</td>
<td>3,072,441</td>
<td>234,370,166</td>
</tr>
<tr>
<td>loc-gowalla</td>
<td>196,591</td>
<td>1,900,654</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>4,843,953</td>
<td>85,691,368</td>
</tr>
<tr>
<td>soc-sign-epinions</td>
<td>119,130</td>
<td>1,408,534</td>
</tr>
<tr>
<td>web-Google</td>
<td>855,802</td>
<td>8,582,704</td>
</tr>
<tr>
<td>web-NotreDame</td>
<td>325,729</td>
<td>2,180,216</td>
</tr>
<tr>
<td>wiki-Talk</td>
<td>2,388,953</td>
<td>9,313,364</td>
</tr>
</tbody>
</table>

• Unable to run [Shi 2011] due to memory constraints except one graph (for which our sequential CPU code is faster than their best GPU implementation)
Comparison of GPU-based Variants

![Comparison of GPU-based Variants](image)

Faster Centrality Computations on GPUs

GTC 2013

HPC Lab

bmi.osu.edu/hpc

Wexner Medical Center
Degree-1 Removal and Ordering (GPU)

![Speedup chart showing performance comparison between GPU stride + deg1 and GPU stride + deg1 + ord across various datasets.](chart.png)
Comparison of CPU (Nehalem), GPU (Fermi) and Heterogeneous Architecture

- **CPU 8 threads + deg1 + ord**
- **GPU virtual + deg1 + ord**
- **GPU stride + deg1 + ord**
- **Heterogeneous virtual + deg1 + ord**
- **Heterogeneous stride + deg1 + ord**

**Speedup wrt CPU 1 thread**

- **amazon001**
- **comorkut**
- **loc-gowalla**
- **soc-livejournal**
- **soc-sign-opinions**
- **web-Google**
- **web-NotreDame**
- **wiki-Talk**
Comparison of CPU (Nehalem), GPU (Fermi) and Heterogeneous Architecture

reduces from ~4 months to ~12 days
Comparison of CPU (Nehalem), GPU (Fermi) and Heterogeneous Architecture

- Reduces from ~4 months to ~12 days
- Speedup of 104
Kepler Experiments

- One quad-core Intel i7 CPU (Sandy Bridge)
  - clocked at 3.20Ghz and have 24GB of memory
  - 32kB L1, 256kB L2, 8MB L3 cache (L3 is shared by 4-core)
- Equipped with an NVIDIA Tesla K20 (Kepler)
  - 4.8GB of global memory, 13 multiprocessors each have 192 CUDA cores (2496 in total)
  - CUDA cores are clocked at 0.71GHz and the memory is clocked at 2.6GHz. ECC is on
Comparison of Fermi, Kepler and Heterogeneous Architecture

Speedup wrt SandyBridge 1 thread

- 4-Core SandyBridge + deg1 + ord
- Fermi stride + deg1 + ord
- Kepler stride + deg1 + ord
- Kepler stride + 3-Core SandyBridge + deg1 + ord

Comparison of Fermi, Kepler and Heterogeneous Architecture

Faster Centrality Computations on GPUs
Comparison of Fermi, Kepler and Heterogeneous Architecture

Speedup wrt SandyBridge 1 thread

- 4-Core SandyBridge + deg1 + ord
- Fermi stride + deg1 + ord
- Kepler stride + deg1 + ord
- Kepler stride + 3-Core SandyBridge + deg1 + ord

Reduces to ~10 days
Comparison of Fermi, Kepler and Heterogeneous Architecture

Kepler is 84% faster than Fermi

reduces to ~10 days
Conclusions

• Various techniques for faster betweenness centrality computation are investigated
  • Virtual vertices work quite well and solves load-balancing issue
  • Stride-CSR representation is very useful
  • Degree-1 removal and graph ordering are effective
  • Utilizing heterogeneous architecture gives much higher speedups

• BC computation is reduced from ~4 months to ~10 days
• Speedups up to 104 are observed
Future Work

- A more detailed profiling can yield new improvements
- Efficiency of GPU algorithms depends on graph diameter, which is usually small in social networks
  - We’ll investigate GPU solutions on large-diameter graphs, like road networks
- There are other sequential techniques to compress and shatter graphs for faster BC [Sariyuce et al. SDM’13]
  - We’ll incorporate them to heterogeneous architectures
Thanks

- For more information
  - Email umit@bmi.osu.edu
  - Visit http://bmi.osu.edu/~umit or http://bmi.osu.edu/hpc

- Research at the HPC Lab is supported by

  ![NSF](image1)
  ![NIST](image2)
  ![NIH](image3)
  ![NCI](image4)
  ![DAGSI](image5)
  ![Sandia National Laboratories](image6)
  ![INL](image7)
  ![NVIDIA](image8)