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
Large Networks and Centrality

- 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 $s$

```
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)
      $d[w] \leftarrow d[v] + 1$
    else
      $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
      P[w].push(v)
```

> 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
      $bc[w] \leftarrow bc[w] + \delta[w]$
```
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
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
- **vmap** array maps the virtual vertices to original vertices

(a) A toy graph $G$

(b) CSR representation of $G$

(d) Virtual-CSR representation of $G$

---

In vertex-based parallelism, all the edges of a single vertex are processed by a single thread. The pseudocode of format for graph storage. Figures show these cases for.

For each if vertex $v$ of graph $G$ 2. To store the predecessor information, a special predecessor list $P$ (b) CSR representation of $G$, its COO representation (c), its virtual-CSR representation (d), and stride-CSR representation (e). In the figures, (a) A toy graph $G$, (b) CSR representation of $G$, (c) COO representation of $G$, (d) Virtual-CSR representation of $G$. 

Hence, in vertex-based parallelism a single atomic operation is the number of vertices, i.e., $n$. Sequential BC will be a prefix step and the corresponding graph storage scheme. To ease understanding that the next frontier will not be empty and will be one of its neighbors. There can be three sets

Let $Q$ be the empty list. And for each level, the algorithms initialize a GPU kernel to visit the vertices/edges on that level.

BC computations: computing betweenness centrality by using GPUs. In the 3. BETWEENNESS CENTRALITY ON GPU

As mentioned above, there are two existing studies on computing betweenness centrality. One is the number of vertices, and the other is the number of entries it has. Considering the size memory accesses, the former uses the compressed sparse row (CSR) format, and the latter uses the coordinate (COO) format for graph storage. Figures show these cases for.

Algorithm 1: http://bioinfo.vanderbilt.edu/gpu-fan/
Parallel BC with Virtualization: 1st phase

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

\begin{algorithm}
\begin{algorithmic}
\STATE $\ell \leftarrow 0$
\STATE >Forward phase
\WHILE {$\text{cont} = \text{true}$}
\STATE $\text{cont} \leftarrow \text{false}$
\STATE >Forward-step kernel
\FOR {each virtual vertex $u_{\text{vir}}$ in parallel}
\STATE $u \leftarrow \text{vmap}[u_{\text{vir}}]$
\IF {$d[u] = \ell$}
\STATE >for each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$
\FOR {each $v$}
\IF {$d[v] = -1$}
\STATE $d[v] \leftarrow \ell + 1$, $\text{cont} \leftarrow \text{true}$
\ENDIF
\ENDIF
\STATE $\ell \leftarrow \ell + 1$
\ENDFOR
\ENDFOR
\ENDWHILE
\end{algorithmic}
\end{algorithm}

original vertex ID
Parallel BC with Virtualization: 1st phase

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

- NO predecessor array, NO queue, only level information
- the avg ratio of active threads in a warp is higher
- Less thread divergence

3

\[ \ell \leftarrow \ell + 1 \]

if \( d[v] = \ell + 1 \) then \( \sigma[v]^{\text{atomic}} \leftarrow \sigma[v] + \sigma[u] \)
Parallel BC with Virtualization: 2nd phase

▷Backward phase
while $\ell > 1$ do
  $\ell \leftarrow \ell - 1$

▷Backward-step kernel
for each virtual vertex $u_{vir}$ in parallel do
  $u \leftarrow vmap[u_{vir}]$
  if $d[u] = \ell$ then
    $sum \leftarrow 0$
    for each $v \in \Gamma_{vir}(u_{vir})$ do
      if $d[v] = \ell + 1$ then $sum \leftarrow sum + \delta[v]$
      $\delta[u] \leftarrow \delta[u] + sum$
  
for each $v \in \Gamma_{vir}(u_{vir})$ do
Parallel BC with Virtualization: 2nd phase

▷ Backward phase

while \( \ell > 1 \) do

\( \ell \leftarrow \ell - 1 \)

▷ Backward-step kernel

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

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

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

\( \text{sum} \leftarrow 0 \)

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

\( \text{if } d[v] = \ell + 1 \text{ then } \text{sum} \leftarrow \text{sum} + \delta[v] \)

\( \delta[u] \leftarrow \delta[u] + \text{sum} \)

for each \( v \in \Gamma_{vir}(u_{vir}) \) do
Parallel BC with Virtualization: 2nd phase

\begin{algorithm}
\caption{Forward phase}
\begin{algorithmic}
\State \textbf{Forward phase}
\While {\( \ell > 1 \)}
\State \( \ell \leftarrow \ell - 1 \)
\EndWhile
\State \textbf{Backward-step kernel}
\For {\textbf{each virtual vertex} \( u_{vir} \) \textbf{in parallel}}
\State \( u \leftarrow vmap[u_{vir}] \)
\If {\( d[u] = \ell \)}
\State \( \text{sum} \leftarrow 0 \)
\For {\textbf{each} \( v \in \Gamma_{vir}(u_{vir}) \)}
\If {\( d[v] = \ell + 1 \)}
\State \( \text{sum} \leftarrow \text{sum} + \delta[v] \)
\EndIf
\EndFor
\State \( \delta[u] \xleftarrow{\text{atomic}} \delta[u] + \text{sum} \)
\EndIf
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

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

\begin{align*}
\text{Backward phase} \\
\hspace{1cm} \text{while } \ell > 1 \text{ do} \\
\hspace{2cm} \ell \leftarrow \ell - 1 \\
\hspace{2cm} \text{Backward-step kernel} \\
\hspace{3cm} \text{for each virtual vertex } u_{\text{vir}} \text{ in parallel do} \\
\hspace{4cm} u \leftarrow \text{vmap}[u_{\text{vir}}] \\
\hspace{4cm} \text{if } d[u] = \ell \text{ then} \\
\hspace{5cm} \text{sum} \leftarrow 0 \\
\hspace{5cm} \text{for each } v \in \Gamma_{\text{vir}}(u_{\text{vir}}) \text{ do} \\
\hspace{6cm} \text{if } d[v] = \ell + 1 \text{ then } \text{sum} \leftarrow \text{sum} + \delta[v] \\
\hspace{5cm} \delta[u]^{\text{atomic}} \leftarrow \delta[u] + \text{sum} \\
\text{for each } v \in \Gamma_{\text{vir}}(u_{\text{vir}}) \text{ do}
\end{align*}

neighbors of virtual vertex are visited
original vertex ID
“sum” is used to reduce number of atomic operations
Parallel BC with Virtualization: 2\textsuperscript{nd} phase

\begin{algorithmic}
\STATE\hspace{1em}\textbf{Backward phase}
\WHILE{$\ell > 1$}
\STATE{$\ell \leftarrow \ell - 1$}
\STATE\hspace{1em}\textbf{Backward-step kernel}
\FOR{each virtual vertex $u_{\text{vir}}$ in parallel}
\STATE{$u \leftarrow \text{vmap}[u_{\text{vir}}]$}
\IF{$d[u] = \ell$}
\STATE{$\text{sum} \leftarrow 0$}
\FOR{each $v \in \Gamma_{\text{vir}}(u_{\text{vir}})$}
\IF{$d[v] = \ell + 1$}
\STATE{$\text{sum} \leftarrow \text{sum} + \delta[v]$}
\ENDIF
\ENDIF
\ENDFOR
\STATE{$\delta[u] \text{atomic} \leftarrow \delta[u] + \text{sum}$}
\ENDFOR
\ENDFOR
\ENDWHILE
\end{algorithmic}


**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

---

**Figure 1: A toy graph**

---

**Algorithm 1:**

```
for all $v \in V$
    do
        $d[v] = 0$
        $P[v] = 0$
        $Q[v] = \emptyset$
    end for

$s \leftarrow S_1$
while $s$ is not empty
    do
        $v \leftarrow s[0]$
        $s \leftarrow s[1]$
        if $v$ is not empty
            then
                for all $w \in V$
                    do
                        if $w$ is a successor of $v$
                            then
                                if $d[w] > d[v] + 1$
                                    do
                                        $d[w] \leftarrow d[v] + 1$
                                        $P[w] \leftarrow v$
                                        $Q[w] \leftarrow Q[w] \cup \{v\}$
                                    end if
                                end if
                            end for
                        end if
                    end for
                end if
            end if
        end if
    end while
```

---

**Table:**

<table>
<thead>
<tr>
<th>vmap</th>
<th>1 2 2 3 4 5 6 7 8 9 10 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>vptrs</td>
<td>1 2 6 10 12 15 18 21 24 27 29 33 35</td>
</tr>
<tr>
<td>adjs</td>
<td>2 1 3 4 5 6 7 8 9 ... 2 10 4 5 6 7 8 9</td>
</tr>
<tr>
<td>offset</td>
<td>0 0 1 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>vmap</td>
<td>1 2 2 3 4 5 6 7 8 9 10 10</td>
</tr>
<tr>
<td>nvir</td>
<td>1 2 1 1 1 1 1 1 1 2</td>
</tr>
<tr>
<td>ptrs</td>
<td>1 2 10 12 15 18 21 24 27 29 35</td>
</tr>
<tr>
<td>adjs</td>
<td>2 1 3 4 5 6 7 8 9 ... 2 10 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

---

**Memory Access**

- **(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
Heterogeneous Computing

- 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

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)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Original</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>amazon0601</td>
<td>403,364</td>
</tr>
<tr>
<td>com-orkut</td>
<td>3,072,441</td>
</tr>
<tr>
<td>loc-gowalla</td>
<td>196,591</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>4,843,953</td>
</tr>
<tr>
<td>soc-sign-epinions</td>
<td>119,130</td>
</tr>
<tr>
<td>web-Google</td>
<td>855,802</td>
</tr>
<tr>
<td>web-NotreDame</td>
<td>325,729</td>
</tr>
<tr>
<td>wiki-Talk</td>
<td>2,388,953</td>
</tr>
</tbody>
</table>
Comparison of GPU-based Variants

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GPU vertex</th>
<th>GPU edge</th>
<th>GPU virtual</th>
<th>GPU stride</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon0601</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>com-orkut</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>loc-gowalla</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>soc-sign-opinion</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>web-Google</td>
<td>10</td>
<td>8</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>web-NotreDame</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>wiki-Talk</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>
Degree-1 Removal and Ordering (GPU)

![Graph showing speedup with respect to GPU stride for various datasets.](image)

- Blue bars represent GPU stride + deg1.
- Red bars represent GPU stride + deg1 + ord.

Datasets include:
- amazon0601
- com-orkut
- loc-gowalla
- soc-LiveJournal
- soc-sign-opinions
- web-Google
- web-NotreDame
- wiki-Talk

GTC 2013

Faster Centrality Computations on GPUs
Comparison of CPU (Nehalem), GPU (Fermi) and Heterogeneous Architecture

![Comparison of CPU (Nehalem), GPU (Fermi) and Heterogeneous Architecture](image)

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

Speedup wrt CPU 1 thread

- CPU 8 threads + deg1 + ord
- GPU virtual + deg1 + ord
- GPU stride + deg1 + ord
- Heterogenous virtual + deg1 + ord
- Heterogenous stride + deg1 + ord

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

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

Graph showing speedup with respect to SandyBridge 1 thread for various datasets and configurations.
Comparison of Fermi, Kepler and Heterogeneous Architecture

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