TurboBC: A Memory Efficient and Scalable GPU Based Betweenness Centrality (BC) Algorithm in the Language of Linear Algebra

Oswaldo Artiles and Fahad Saeed
TurboBC

The first implementation of a set of memory efficient Brandes’ BC algorithms in the language of linear algebra, applicable to unweighted sparse graphs.

- Good performance
- High scalability
TurboBC
Optimization strategies

Exploiting the sparsity of the frontier and output vector of the top-down BFS algorithms

Improving the performance of the BC algorithms
TurboBC Optimization strategies

Minimizing the number of arrays on the GPU global memory

Reducing the memory footprint

Increasing the memory efficiency and the scalability of the TurboBC algorithms
TurboBC applicable to:

Sparse unweighted graphs represented by binary sparse adjacency matrices in compressed sparse formats.
TurboBC
Sparse Compressed Formats

Sparse Adjacency Matrix

idx: 1 2 3 4 5 6 7
CSC format:
row: 2 3 1 4 5 1 4
offset: 1 2 3 6 7 8
COOC format:
row: 2 3 1 4 5 1 4
col: 1 2 3 3 4 5
TurboBC
Regular and Irregular Graphs
Tweenness centrality of a vertex v of a graph G

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

One-sided dependences

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$$

Brandes’ recurrence relation to compute the one-sided dependences

$$\delta_s(v) = \sum_{w: d(s, w) = d(s, v) + 1} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w))$$
TurboBC
Two-stages algorithm

Forward stage: Top-down BFS algorithm to compute the shortest paths/vertex

Backward stage: computation of one-sided dependences and update BC/vertex
**TurboBC**

**Forward stage: Top-down BFS algorithm**

```plaintext
1: for \( s \leftarrow 1, n \) do
2:     while \( f > 0 \) do
3:         \( f_t \leftarrow A^T f \)
4:         if \( \exists \sigma(i) = 0 \) then
5:             \( f(i) \leftarrow f_t(i) \)
6:         end if
7:         if \( \exists f(i)! = 0 \) then
8:             \( S(i) \leftarrow d \)
9:             \( \sigma(i) \leftarrow \sigma(i) + f(i) \)
10:            \( c \leftarrow 1 \)
11:        end if
12:     end while
13: end for
```
TurboBC

Backward stage: computation of one-sided dependences and update BC/vertex

```plaintext
1: for s ← 1, n do  # s: source vertex of BFS tree
2:     while d > 1 do  # one-sided dependences vector stage
3:         if S(i) == d and σ(i) > 0 then
4:             δ_u(i) ← (1.0 + δ(i)) / σ(i)
5:         end if
6:         δ_ut ← A^T δ_u
7:         if S(i) == d - 1 then
8:             δ(i) ← δ(i) + δ_ut(i) × σ(i)
9:         end if
10:     d ← d - 1
11: end while
12: for v ← 1, n do  # updating betweenness centrality bc
13:     if v ≠ s then
14:         bc(v) ← bc(v) + δ(v)
15:     end if
16: end for
```
TurboBC
Sparse Matrix-Vector Multiplication
Regular Graphs

One thread per edge
COOC-scalar
regular graphs

Sequential $f_t \leftarrow A^T f$ scalar operation
COOC format

1: for $k \rightarrow 1, nnz$ do
2:   if $f(row_A(k)) > 0$ then
3:     $f_t(col(k)) \leftarrow f_t(col(k)) + f(row(k))$

CSC format

1: for $v \rightarrow 1, n$ do
2:   if $\sigma(v) == 0$ then
3:     sum $\leftarrow 0$
4:     start $\leftarrow$ offset$(v)$
5:     end $\leftarrow$ offset$(v + 1) - 1$
6:     for $k \rightarrow start, end$ do
7:       sum $\leftarrow$ sum + $f(row(k))$
8:     end for
9:   if sum $> 0$ then
10:      $f_t(v) \leftarrow$ sum

One thread per vertex
CSC-scalar
Regular graphs
TurboBC
Sparse Matrix-Vector Multiplication
Warp Divergence

Irregular graphs: Nodes with higher degrees

Load unbalance in warps

Warp divergence
TurboBC
Sparse Matrix-Vector Multiplication
Irregular Graphs

One Warp (32 Threads) per Vertex
Parallel reduction
CSC-vector

1: procedure veCSC-MVSP-KERNEL(offset, row, f)
2: if σ(col) == 0 then
3:   start ← offset(col)
4:   end ← offset(col + threadLane_id)
5: while icp < end do
6:   sum ← sum + f(rowA(icp))
7:   icp ← icp + threadsPerWarp
8: end while
9: off ← threadsPerWarp/2
10: while off > 0 do
11:   sum ← sum + shfl – down – sync(off)
12:   off ← off2
13: end while
14: if threadLane_id == 0 then
15:   f_i(warp_id) ← sum
16: end if
Thirty-three graphs represented by sparse adjacency matrices from the SuiteSparse Matrix Collection, with up to 1.9 billion edges and 214 million vertices.

*gunrock*: High-performance GPU-based graph analytic library

*ligra*: High-performance CPU-based graph analytic library
TurboBC

GPU Memory Efficiency: TurboBC vs gunrock

a) GPU memory usage
b) Global Memory Load Throughput (GB/s)
c) Performance (MTEPs)
TurboBC
Experimental Results for Regular Graphs

- Ten graphs
- Up to 470 MTEPs (Million transverse edges/second)
TurboBC
Experimental Results for Regular Graphs

- Ten graphs
- Up to 1000 MTEPs (Million transverse edges/second)
TurboBC
Experimental Results for Irregular Graphs

- Nine graphs
- TurboBC CSC-vector
- Up to 18470 MTEPs (Million transverse edges/second)
TurboBC
Experimental Results for Big Graphs

<table>
<thead>
<tr>
<th>File</th>
<th>n×10^6</th>
<th>m×10^6</th>
<th>degree(max/μ/σ)</th>
<th>d</th>
<th>scf</th>
<th>runtime(s)</th>
<th>MTEPs</th>
<th>(sequential)x</th>
<th>(gunrock)x</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmer-V1r(U)</td>
<td>214</td>
<td>465</td>
<td>8/2/1</td>
<td>324</td>
<td>2</td>
<td>14.3</td>
<td>33</td>
<td>94.5</td>
<td>OOM</td>
</tr>
<tr>
<td>it-2004(D)</td>
<td>42</td>
<td>1151</td>
<td>9964/28/67</td>
<td>50</td>
<td>543</td>
<td>3.1</td>
<td>371</td>
<td>39.5</td>
<td>OOM</td>
</tr>
<tr>
<td>GAP-twitter(D)</td>
<td>62</td>
<td>1469</td>
<td>3×10^6/24/1990</td>
<td>15</td>
<td>126</td>
<td>7.3</td>
<td>201</td>
<td>50.4</td>
<td>OOM</td>
</tr>
<tr>
<td>sk-2005(D)</td>
<td>51</td>
<td>1950</td>
<td>12870/39/78</td>
<td>54</td>
<td>1262</td>
<td>6.8</td>
<td>287</td>
<td>30.5</td>
<td>OOM</td>
</tr>
</tbody>
</table>

The BC algorithms available in the gunrock libraries ran out of memory for these big graphs.

High scalable TurboBC algorithms
TurboBC

Experimental Results Summary
TurboBC main result

A memory efficient and highly scalable first implementation of GPU-based set of Brandes’ BC algorithms in the language of linear algebra.
Thank you

Questions?
TurboBFS
Experimental CPU-GPU Platform

• Linux server with Ubuntu operating system version 16.04.6, 22 Intel Xeon Gold 6152 processors, clock speed 2.1 GHz, and 125 GB of RAM.

• The GPU in this server was a NVIDIA Titan Xp, with 30 SM, 128 cores/SM, maximum clock rate of 1.58 GHz, 12196 MB of global memory, and CUDA version 10.1.243 with CUDA capability of 6.1.
- **Aligned memory access** occurs when the first address of a device memory transaction is an even multiple of 32 bytes for L2 cache or 128 bytes for L1 cache.

- **Coalesced memory access** occurs when all 32 threads in a warp access a contiguous chunk of memory.

- **Aligned coalesced memory access** is ideal because it maximizes global load memory throughput. That is, the addresses requested by all threads in a warp fall within one cache line of 128 bytes. Only a single 128-byte transaction is required by the memory load operation.
When the L1 cache is enabled, three 128-byte memory transactions may be required, resulting in wasted memory bandwidth because some of the bytes loaded are not used.

Misaligned accesses can be verified by collecting information of the Global Memory Load Efficiency (GMLE) metrics.