Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure

Alok Tripathy
What I’ll Show

• Maximal $k$-core algorithm
  – Up to $4X$ faster than previous research
  – Up to $58X$ faster than popular graph libraries

• $k$-core edge decomposition algorithm
  – Up to $8X$ faster than previous research
  – Up to $129X$ faster than popular graph libraries
What I’ll Show

- Maximal $k$-core algorithm
  - Up to $4X$ faster than previous research
  - Up to $58X$ faster than popular graph libraries

- $k$-core edge decomposition algorithm
  - Up to $8X$ faster than previous research
  - Up to $129X$ faster than popular graph libraries
  - Uses a dynamic graph operations
Takeaways

• Algorithms on static graphs can use dynamic graph operations efficiently with the GPU.

• Dynamic graph operations can be computed on a GPU efficiently.
  – Check out the Hornet data structure!
  – https://github.com/hornet-gt/hornet
Motivation

• Two types of graphs
  – Static graphs that don’t change
  – Dynamic graphs that change frequently
    • Edge/vertex insertions/deletions
    • e.g. Facebook, road networks
Motivation

- Two types of graphs
  - Static graphs that don’t change
  - Dynamic graphs that change frequently
    - Edge/vertex insertions/deletions
    - e.g. Facebook, road networks

- Algorithms on static graphs can benefit from dynamic graph operations
Dynamic Operations on Static Graphs

- $k$-truss problem
Dynamic Operations on Static Graphs

- $k$-truss problem
  - Subgraph where all edges belong to at least $k - 2$ triangles
  - Can be extended to maximal $k$-truss

$k = 4$
• *k*-truss problem
  - Subgraph where all edges belong to at least \( k - 2 \) triangles
  - Can be extended to maximal \( k \)-truss
  - Applications: community detection, anomaly detection

\[ k = 4 \]
$k$-truss Algorithm

- $E_m = \text{all edges in } \geq k - 2 \text{ triangles}$

- while $|E_m| > 0$
  - delete $E_m$ from G
  - update triangles in G
  - $E_m = \text{all edges in } \geq k - 2 \text{ triangles}$
Takeaways

• Algorithms on static graphs can use dynamic graph operations efficiently with the GPU.

• Dynamic graph operations can be computed on a GPU efficiently.
  – Check out the Hornet data structure!
  – https://github.com/hornet-gt/hornet
## Widely used graph data structures

<table>
<thead>
<tr>
<th>Names</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Adjacency Matrix</td>
<td>• Supports updates</td>
<td>• Poor locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Massive storage requirements</td>
</tr>
<tr>
<td>Linked lists</td>
<td>• Flexible</td>
<td>• Poor locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Limited parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Allocation time is costly</td>
</tr>
<tr>
<td>COO (Edge list) - unsorted</td>
<td>• Has some flexibility</td>
<td>• Poor locality</td>
</tr>
<tr>
<td></td>
<td>• Updates are simple</td>
<td>• Stores both the source and destination</td>
</tr>
<tr>
<td></td>
<td>• Lots of parallelism</td>
<td></td>
</tr>
<tr>
<td>CSR</td>
<td>• Uses exact amount of memory</td>
<td>• Inflexible</td>
</tr>
<tr>
<td></td>
<td>• Good locality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Lots of parallelism</td>
<td></td>
</tr>
</tbody>
</table>

These data structures don’t cut it
Compressed Sparse Row (CSR)

Pros:
- Uses precise storage requirements
- Great locality
  - Good for GPUs
- Handful of arrays
  - Simple to use and manage

Cons:
- Inflexible.
- Network growth unsupported
- Topology changes unsupported
- Property graphs not supported
Hornet – A High Level View

USER-INTERFACE

Over-allocated space

Dest Value

Vertex Id

Used

Pointer

Oded Green, Alok Tripathy, GTC 2019
Hornet in Detail

USER-INTERFACE
Vertex Id: 0 1 2 3 4 5 6 7
Used (#Neighbors/nnz): 2 2 3 2 2 2 1 0
Pointer:

MEMORY MANAGER

BA<sub>0,1</sub>, bsize=1
BA<sub>1,1</sub>, bsize=2
BA<sub>1,2</sub>, bsize=2
BA<sub>2,1</sub>, bsize=4

Bit status
Vec-Tree

Over-allocated space for vertex insertions
Over-allocated space for power-of-two rule

Dest./Col. Weight

Oded Green, Alok Tripathy, GTC 2019
Hornet Insertion

(b) The updated graph.
Hornet Insertion Pseudocode

parallel for (u, v) in batch
  - if u’s block is too full
    - allocate a new block
    - queue.add(u)

parallel for v in queue
  - copy adjacency list to new block

parallel for (u, v) in batch
  - add (u, v) to u’s block
Insertion Rates

• Supports over 150M updates per second

• Hornet
  – $4X - 10X$ faster than cuSTINGER
  – Does not have performance dip like cuSTINGER

• Scalable growth in update rate
Takeaways

• Algorithms on static graphs can use dynamic graph operations efficiently with the GPU.

• Dynamic graph operations can be computed on a GPU efficiently.
  – Check out the Hornet data structure!
  – https://github.com/hornet-gt/hornet
Motivation

• Current idea:
  – Dynamic graph operations are only for dynamic graphs, not static graphs.
    • Very expensive
    • Why bother?
Motivation

• Current idea:
  – Dynamic graph operations are only for dynamic graphs, not static graphs.
    • Very expensive
    • Why bother?

• New idea: Algorithms on static graphs can benefit from dynamic graph operations
  – If we can efficiently parallelize operations
What I’ll Show

• 3 static graph algorithms
  – All 3 leverage NVIDIA P100 GPUs.
    • 2 beat the state-of-the-art
    • 1 does not (does not have good GPU utilization)
Algorithms

- Old maximal $k$-core algorithm
- New maximal $k$-core algorithm
- $k$-core edge decomposition
Algorithms

• Old maximal $k$-core algorithm 😞
• New maximal $k$-core algorithm
• $k$-core edge decomposition
Maximal $k$-core Definitions

- **$k$-core**
  - Maximal subgraph where all vertices have degree at least $k$

$k = 2$
Maximal $k$-core Definitions

- **$k$-core**
  - Maximal subgraph where all vertices have degree at least $k$

- **Maximal $k$-core**
  - Largest $k$ such that $k$-core exists in graph
Maximal $k$-core Definitions

• $k$-core
  - Maximal subgraph where all vertices have degree at least $k$

• Maximal $k$-core
  - Largest $k$ such that $k$-core exists in graph

• Applications: visualization, community detection
Maximal $k$-core High-Level

$\text{peel} = 0$

while vertices exist in $G$

- delete all vertices with degree $\leq \text{peel}$

- if there aren’t any
  - increment $\text{peel}$

$\text{peel} = 1$

Alok Tripathy, GTC 2019
Maximal $k$-core High-Level

$\text{peel} = 0$
while vertices exist in $G$
- delete all vertices with degree $\leq \text{peel}$
- if there aren’t any
- increment $\text{peel}$

$\text{peel} = 2$
Maximal $k$-core High-Level

- $peel = 0$
  while vertices exist in $G$
  - delete all vertices with degree $\leq peel$
  - if there aren’t any
    - increment $peel$

$peel = 3$
Old Maximal $k$-core Algorithm

$peel = 0$
while vertices exist in $G$
   - reset colors
   - color all vertices with degree $\leq peel$
      - if #coloredvertices $> 0$
         - delete colored vertices
         - delete incident edges
         - insert vertices in $G$
         - insert edges in $\hat{G}$
      - else
         - increment $peel$
Old Maximal $k$-core Code

```c
while (nv > 0) {
    forAllVertices(hornet, SetColor { vertex_color });
    forAllVertices(hornet, CheckDeg { vqueue, peel_vqueue, vertex_pres, vertex_color, peel });

    vqueue.swap();
    nv -= vqueue.size();

    if (vqueue.size() > 0) {
        cudaMemset(hd().counter);

        forAllEdges(hornet, vqueue, PeelVertices { hd, vertex_color }, load_balancing);
        cudaMemcpy(&size, hd().counter, sizeof(int), cudaMemcpyDeviceToHost);
        if (size > 0) {
            oper_bidirect_batch(hornet, hd().src, hd().dst, size, DELETE);
            oper_bidirect_batch(h_copy, hd().src, hd().dst, size, INSERT);
        }
        *ne -= size;
    }
    vqueue.clear();
} else {
    peel++;
    peel_vqueue.swap();
}
}
*max_peel = peel;
```
Compared Against

- **ParK**
  - parallel $k$-core algorithm; IEEE BigData 2014
  - Some parallelism
  - No dynamic graph operations

- **igraph**
  - network analysis toolkit
  - Sequential
  - No dynamic graph operations

- Both run on Intel Xeon E5-2695; 36 cores, 72 threads
Old Maximal $k$-core Results

- Our algorithm is sometimes better than igraph.
- Our algorithm never beats ParK.

- Why are we so slow?

| Name                  | $|V|$     | $|E|$   | Our algorithm | ParK | igraph |
|-----------------------|---------|--------|---------------|------|--------|
| dblp – author         | 5.5M    | 8.6M   | 2.2X          | 15X  | 1X     |
| patentcite            | 3.8M    | 16.5M  | 1.3X          | 15X  | 1X     |
| soc – LiveJournal1    | 4.8M    | 42.9M  | OOM           | 11.3X| 1X     |
| soc – pokec – relationships | 1.6M   | 22.3M  | 0.6X          | 16.6X| 1X     |
| trackers              | 27.7M   | 140.6M | OOM           | 6.8X | 1X     |
| wikipedia – link – de | 3.2M    | 65.8M  | OOM           | 5.1X | 1X     |

Alok Tripathy, GTC 2019
GPU Utilization

Percentage of GPU Utilized vs. Fraction of execution completed

Alok Tripathy, GTC 2019
GPU Utilization / Batch Size

![Graph showing GPU utilization and batch size relationship](image)

Update Rate (edges per second)

- in-2004
- soc-LiveJournal1
- cage15
- kron_g500-logn21
Algorithms

- Old maximal $k$-core algorithm 😞
- New maximal $k$-core algorithm
- $k$-core edge decomposition
Algorithms

- Old maximal $k$-core algorithm 😞
- New maximal $k$-core algorithm 😊
- $k$-core edge decomposition
New Maximal $k$-core Algorithm

• Flag vertices instead of deleting them.

while not every vertex is flagged

- flag all vertices with degree $\leq peel$

- if there aren’t any
  - increment $peel$

- else
  - for each flagged vertex $v$
    - for each neighbor of $v$
      - decrement neighbor’s degree
New Maximal $k$-core Code

```c
int n_active = active_queue.size();
uint32_t peel = 0;

while (n_active > 0) {
    forAllVertices(hornet, active_queue,
        PeelVerticesNew { vertex_pres, deg, peel, peel_queue, iter_queue} );
    iter_queue.swap();

    n_active -= iter_queue.size();

    if (iter_queue.size() == 0) {
        peel++;
        peel_queue.swap();
        if (n_active > 0) {
            forAllVertices(hornet, active_queue, RemovePres { vertex_pres });
        }
    } else {
        forAllEdges(hornet, iter_queue, DecrementDegree { deg }, load_balancing);
    }
}
```
New Maximal $k$-core Results

- Our algorithm always beats igraph.
- Our algorithm is sometimes better than ParK.
  - At best, 3.9X faster
  - At worst, 4.3X slower
- Learned that batch size affected performance.

| Name                  | $|V|$ | $|E|$ | Our algorithm | ParK | igraph |
|-----------------------|-----|-----|---------------|------|--------|
| dblp – author         | 5.5M| 8.6M| 58X           | 15X  | 1X     |
| patentcite            | 3.8M| 16.5M| 26X           | 15X  | 1X     |
| soc – LiveJournal1    | 4.8M| 42.9M| 7.4X          | 11.3X| 1X     |
| soc – pokec – relationships | 1.6M| 22.3M| 15X           | 16.6X| 1X     |
| trackers              | 27.7M| 140.6M| 1.6X         | 6.8X | 1X     |
Algorithms

- Old maximal $k$-core algorithm 😞
- New maximal $k$-core algorithm 😊
- $k$-core edge decomposition
Algorithms

• Old maximal $k$-core algorithm 😞
• New maximal $k$-core algorithm 😊
• $k$-core edge decomposition 😊
**$k$-core Decomp. Definitions**

- **$k$-core edge decomposition**
  - For each edge, what is the largest $k$-core that edge belongs to?

![Diagram](image-url)
while vertices exist in G

- find the maximal k-core in G

- mark all edges in k-core with value k

- delete k-core from G
\(k\)-core Decomp. Code

```c
while (peel_edges < ne) {
    uint32_t max_peel = 0;
    int batch_size = 0;

    maximal_kcore(hornet, hd_data, peel_vqueue, active_queue, iter_queue,
                  load_balancing, vertex_deg, vertex_pres, &max_peel, &batch_size);

    if (batch_size > 0) {
        cudaMemcpy(src + peel_edges, hd_data().src,
                    batch_size * sizeof(vid_t), cudaMemcpyDeviceToHost);

        cudaMemcpy(dst + peel_edges, hd_data().dst,
                    batch_size * sizeof(vid_t), cudaMemcpyDeviceToHost);

        #pragma omp parallel for
        for (uint32_t i = 0; i < batch_size; i++) {
            peel[peel_edges + i] = max_peel;
        }

        peel_edges += batch_size;
    }
}

oper_bidirect_batch(hornet, hd_data().src, hd_data().dst, batch_size, DELETE);
```
Compared Against

- ParK Extension
  - parallel $k$-core algorithm; IEEE BigData 2014
  - Some parallelism
  - No dynamic graph operations – vertex flagging

- igraph Extension
  - network analysis toolkit
  - Sequential
    - Uses edge deletions

- Both run on Intel Xeon E5-2695; 36 cores, 72 threads
**$k$-core Decompo. Results**

- Our algorithm always beats igraph
- Our algorithm always beats ParK ($1.2X - 7.8X$).
  - Usually $\sim 2X$ faster
- Our algorithm uses dynamic graph operations
  - And effectively uses the GPU

| Name                  | $|V|$ | $|E|$ | Our algorithm | ParK  | igraph |
|-----------------------|------|------|---------------|-------|--------|
| dblp – author         | 5.5M | 8.6M | 129.2X        | 51.5X | 1X     |
| patentcite            | 3.8M | 16.5M| 63.8X         | 25X   | 1X     |
| soc – LiveJournal1    | 4.8M | 42.9M| 25.9X         | 3.3X  | 1X     |
| soc – pokc – relationships | 1.6M | 22.3M| 85.9X         | 36.3X | 1X     |
| trackers              | 27.7M| 140.6M| 4.7X         | 4.1X  | 1X     |
\( k \)-core Decomp. GPU Utilization

Percentage of GPU Utilized vs Fraction of execution completed
Decomp. vs. Slow Maximal $k$-core
Conclusion

• Dynamic graph operations can be computed on a GPU efficiently.

• Current idea:
  – Dynamic graph operations are only for dynamic graphs, not static graphs

• New idea: Static graph algorithms can benefit from dynamic graph operations
  – If we can efficiently utilize the system
Takeaway

• Consider dynamic graph operations when you implement graph algorithms
  – Even if the graph doesn’t change over time.
Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure

Alok Tripathy
Georgia Tech
atripathy8@gatech.edu
@alokpathy
www.aloktripathy.me

Fred Hohman
Georgia Tech
fredhohman@gatech.edu
@fredhohman
fredhohman.com

Polo Chau
Georgia Tech
polo@gatech.edu
@PoloChau
cc.gatech.edu/~dchau/

Oded Green
Georgia Tech/NVIDIA
ogreen@gatech.edu
@OdedGreen

- _k_-core Paper: Proceedings of IEEE BigData 2018
- _k_-truss, Hornet Paper: Proceedings of IEEE HPEC 2017/18
- Code: https://github.com/hornet-gt/hornet

Thank you
Backup slides
Performance

• Compared against
  - ParK: parallel \( k \)-core algorithm; BigData 2014
  - igraph: network analysis toolkit

• Dynamic graph data structure
  - Hornet, GPU-based

• Systems used
  - Our algorithms: NVIDIA P100
  - ParK, igraph: Intel Xeon E5-2695; 36 cores, 72 threads
    • igraph is sequential
Performance

• Compared against
  – Wang & Cheng: sequential algorithm for finding $k$-truss
  – Graphulo: parallel algorithm for finding $k$-truss

• Dynamic graph data structure
  – cuSTINGER-Delta, GPU-based
    • Evolved into Hornet

• Systems used
  – Our algorithm: NVIDIA P100
  – Wang & Cheng: Intel Core2 dual-core 2.80GHz CPU
  – Graphulo: 2 Intel i7 dual-core
GPU Utilization / Batch Size

Alok Tripathy, GTC 2019
HKS (maximal k-core) results

- ParK: k-core algorithm from IEEE Big Data 2014
- HKS run on NVIDIA P100 with Hornet data structure.

| Name            | $|V|$  | $|E|$   | HKS (sec.) | ParK (sec.) | igraph (sec.) |
|-----------------|------|--------|---------|----------|-------------|---------------|
| dblp – author   | 5.5M | 8.6M   | 0.731   | 0.105   | 1.633       | 1X            |
| patentcite      | 3.8M | 16.5M  | 2.953   | 0.253   | 3.825       | 1X            |
| soc – LiveJournal1 | 4.8M | 42.9M  | OOM     | 0.549   | 6.191       | 1X            |
| soc – pokec – relationships | 1.6M | 22.3M  | 4.331   | 0.155   | 2.586       | 1X            |
| trackers        | 27.7M | 140.6M | OOM     | 3.052   | 20.693      | 1X            |
| wikipedia – link – de | 3.2M | 65.8M  | OOM     | 0.764   | 3.954       | 1X            |
HDS (k-core decomp) results

- ParK: k-core algorithm from IEEE Big Data 2014
- HDS run on NVIDIA P100 with Hornet data structure.

| Name            | |V|  | |E|   | HDS (sec.) | ParK (sec.) | igraph (sec.) |
|-----------------|---------|--------|----------|-------|------------|-------------|--------------|
| dblp – author   | 5.5M    | 8.6M   | 6.184    | 13.3X | 1.595      | 51.5X       | 82.066       |
| patentcite      | 3.8M    | 16.5M  | 91.481   | 3.6X  | 13.294     | 25X         | 331.538      |
| soc – LiveJournal1 | 4.8M  | 42.9M  | OOM      |       | 487.112    | 3.3X        | 1572.985     |
| soc – pokec – relationships | 1.6M | 22.3M  | 50.049  | 4.7X  | 6.488      | 36.3X       | 235.790      |
| trackers        | 27.7M   | 140.6M | OOM      |       | 1148.638   | 4.1X        | 4725.317     |
| wikipedia – link – de | 3.2M | 65.8M  | OOM      |       | 1397.323   | 2.1X        | 3003.166     |

Alok Tripathy, BigData 2018
GPU Utilization

Alok Tripathy, BigData 2018
GPU Utilization / Batch Size
Maximal K-Core Algorithm (HKO)

while there are non-flagged vertices

flag all vertices with degree $\leq peel$

if there aren’t any
  increment $peel$
else
  for each flagged vertex $v$
    for each neighbor of $v$
      decrement neighbor’s degree
Maximal K-Core Algorithm (HKO)

```
peel ← 1
Q ← {}
num_active = |V(G)|

color[v] ← 0 ∀ v ∈ V(G)
deg[v] ← G.deg(v) ∀ v ∈ V(G)

while num_active > 0 do
    V_b ← {}
    parallel for v ∈ V(G) ∧ ! flag[v] do
        if deg[v] ≤ peel then
            flag[v] ← 1
            V_b.enqueue(v)
    end parallel for
    Q ← Q ∪ V_b
    num_active ← num_active − |V_b|

    if |V_b| > 0 then
        parallel for (u, v) : u ∈ V_b, v ∈ adj(u) do
            deg[u] ← deg[u] − 1
            deg[v] ← deg[v] − 1
        end parallel for
    else
        peel ← peel + 1
        Q ← {}
    end if

return (induced_subgraph(G, Q), peel)
```
Maximal K-Core Algorithm 1 (HKS)

```
peel ← 1
Q ← {}
\( \hat{G} \leftarrow (\{\}, \{\}) \)
while \( |V(\hat{G})| > 0 \) do
    color[v] ← 0 ∀v ∈ V(\( \hat{G} \))
    \( V_b \leftarrow \{\} \)
    // Mark vertices with degree ≤ peel
    parallel for v ∈ V(\( \hat{G} \)) do
        if deg[v] ≤ peel then
            color[v] ← 1
            \( V_b\.enqueue(v) \)
    end parallel for
    if |\( V_b \)| > 0 then
        \( E_b \leftarrow \{\} \)
        // Mark edges with at least one marked vertex
        parallel for (u, v) : u ∈ \( V_b \), v ∈ adj(u) do
            if color[u] or color[v] then
                \( E_b\.enqueue((u, v)) \)
        end parallel for
        // Delete these edges from \( \hat{G} \)
        \( G\.delete\.edges(E_b) \)
        \( G\.delete\.vertices(V_b) \)
        // Insert these edges into \( \hat{G} \)
        \( \hat{G}.insert\.vertices(V_b) \)
        \( \hat{G}.insert\.edges(E_b) \)
        Q ← Q ∪ \( V_b \)
    else
        peel ← peel + 1
        Q ← {}
    end if
return (induced_subgraph(\( \hat{G}, Q \)), peel)
```
Maximal K-Core Algorithm 1 (HKS)

\[
\text{peel} \leftarrow 1
\]
\[
Q \leftarrow \{
\}
\]
\[
\tilde{G} \leftarrow (\{
\},\{
\})
\]

while \(|V(G)| > 0\) do
\[
\text{color} [v] \leftarrow 0 \quad \forall v \in V(G)
\]
\[
V_b \leftarrow \{
\}
\]

// Mark vertices with degree \(\leq \text{peel}\)
parallel for \(v \in V(G)\) do
\[
\text{if } \text{deg}[v] \leq \text{peel} \text{ then}
\]
\[
\text{color}[v] \leftarrow 1
\]
\[
V_b.\text{enqueue}(v)
\]
end parallel for

if \(|V_b| > 0\) then
\[
E_b \leftarrow \{
\}
\]

// Mark edges with at least one marked vertex
parallel for \((u, v) : u \in V_b, v \in \text{adj}(u)\) do
\[
\text{if } \text{color}[u] \text{ or color}[v] \text{ then}
\]
\[
E_b.\text{enqueue}((u, v))
\]
end parallel for

// Delete these edges from \(G\)
\[
G.\text{delete}\_\text{edges}(E_b)
\]
\[
G.\text{delete}\_\text{vertices}(V_b)
\]

// Insert these edges into \(\tilde{G}\)
\[
\tilde{G}.\text{insert}\_\text{vertices}(V_b)
\]
\[
\tilde{G}.\text{insert}\_\text{edges}(E_b)
\]
\[
Q \leftarrow Q \cup V_b
\]
else
\[
\text{peel} \leftarrow \text{peel} + 1
\]
\[
Q \leftarrow \{
\}
\]
return \((\text{induced}\_\text{subgraph}(\tilde{G}, Q), \text{peel})\)
Maximal K-Core Algorithm 1 (HKS)

peel ← 1
Q ← {}
\( \tilde{G} \) ← (\{\}, \{\})
while \(|V(G)| > 0\) do
    color[v] ← 0 \( \forall v \in V(G) \)
    \( V_b \) ← {}
    // Mark vertices with degree ≤ peel
    parallel for \( v \in V(G) \) do
        if \( \text{deg}[v] \leq \text{peel} \) then
            color[v] ← 1
            \( V_b \).enqueue(\( v \))
    end parallel for
    if \(|V_b| > 0\) then
        \( E_b \) ← {}
        // Mark edges with at least one marked vertex
        parallel for \((u, v) : u \in V_b, v \in \text{adj}(u)\) do
            if color[u] or color[v] then
                \( E_b \).enqueue((\( u, v \)))
        end parallel for
        // Delete these edges from \( G \)
        \( G \).delete_edges(\( E_b \))
        \( G \).delete_vertices(\( V_b \))
        // Insert these edges into \( \tilde{G} \)
        \( \tilde{G} \).insert_vertices(\( V_b \))
        \( \tilde{G} \).insert_edges(\( E_b \))
        \( Q \) ← \( Q \cup V_b \)
    else
        peel ← peel + 1
        Q ← {}
    end if
return \( \text{induced_subgraph}(\tilde{G}, Q), \text{peel} \)
Maximal K-Core Algorithm 1 (HKS)

```
peel ← 1
Q ← {}
\( \tilde{G} \) ← (\{\}, \{\})
while \(|V(\tilde{G})| > 0\) do
    color[v] ← 0 \(\forall v \in V(\tilde{G})\)
    \( V_b \) ← {}
    // Mark vertices with degree \( \leq \text{peel} \)
    parallel for \( v \in V(\tilde{G}) \) do
        if \( \text{deg}[v] \leq \text{peel} \) then
            color[v] ← 1
            \( V_b.semi.enqueue(v) \)
    end parallel for

    if \(|V_b| > 0\) then
        \( E_b \) ← {}
        // Mark edges with at least one marked vertex
        parallel for \( (u, v) : u \in V_b, v \in \text{adj}(u) \) do
            if color[u] or color[v] then
                \( E_b.semi.enqueue((u, v)) \)
        end parallel for
        // Delete these edges from \( \tilde{G} \)
        \( G.delete\_edges(E_b) \)
        \( G.delete\_vertices(V_b) \)
        // Insert these edges into \( \tilde{G} \)
        \( \tilde{G}.insert\_vertices(V_b) \)
        \( \tilde{G}.insert\_edges(E_b) \)
        \( Q \) ← \( Q \cup V_b \)
    else
        peel ← peel + 1
        Q ← {}
    end if
    \( \tilde{G} \) ← \( \tilde{G} \setminus V_b \)
```

return \((\text{induced}\_\text{subgraph}(\tilde{G}, Q), \text{peel})\)
Maximal K-Core Algorithm 1 (HKS)

peel ← 1
Q ← {}
G ← ({}, {})
while |V(G)| > 0 do
    color[v] ← 0 ∀ v ∈ V(G)
    V_b ← {}
    // Mark vertices with degree ≤ peel
    parallel for v ∈ V(G) do
        if deg[v] ≤ peel then
            color[v] ← 1
            V_b.enqueue(v)
    end parallel for
    if |V_b| > 0 then
        E_b ← {}
        // Mark edges with at least one marked vertex
        parallel for (u, v) : u ∈ V_b, v ∈ adj(u) do
            if color[u] or color[v] then
                E_b.enqueue((u, v))
        end parallel for
        // Delete these edges from G
        G.delete_edges(E_b)
        G.delete_vertices(V_b)
        // Insert these edges into \( \tilde{G} \)
        \( \tilde{G}.insert\_vertices(V_b) \)
        \( \tilde{G}.insert\_edges(E_b) \)
        Q ← Q ∪ V_b
    else
        peel ← peel + 1
        Q ← {}
    return (induced_subgraph(\( \tilde{G} \), Q), peel)
K-Core Decomp. Algorithm 1 (HDS)

\[
\hat{G} \leftarrow (\{\}, \{\})
\]

while \(|V(G)| > 0\) do

// Find maximal \(k\)-core of \(G\)
\(K, k\_num \leftarrow KcoreNum1(G, \hat{G})\)

// Mark edges in the maximal \(k\)-core with the peel number
parallel for \(e \in E(K)\) do

\(peels[e] \leftarrow k\_num\)

end parallel for

// Delete the \(k\)-core edges and vertices
\(\hat{G}.delete\_edges(E(K))\)
\(\hat{G}.delete\_vertices(V(K))\)

swap\((G, \hat{G})\)

return \(peels[]\)
**HKO (maximal k-core) results**

- ParK: k-core algorithm from IEEE Big Data 2014
- HKO run on NVIDIA P100 with Hornet data structure.

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
<th>V</th>
<th></th>
<th></th>
<th>E</th>
<th></th>
<th>HKO (sec.)</th>
<th>ParK (sec.)</th>
<th>igraph (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblp – author</td>
<td>5.5M</td>
<td>8.6M</td>
<td>0.028</td>
<td>0.105</td>
<td>1.633</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15X</td>
<td>15X</td>
<td>1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>patentcite</td>
<td>3.8M</td>
<td>16.5M</td>
<td>0.147</td>
<td>0.253</td>
<td>3.825</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26X</td>
<td>15X</td>
<td>1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>soc – LiveJournal1</td>
<td>4.8M</td>
<td>42.9M</td>
<td>0.838</td>
<td>0.549</td>
<td>6.191</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.4X</td>
<td>11.3X</td>
<td>1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>soc – pokec – relationships</td>
<td>1.6M</td>
<td>22.3M</td>
<td>0.174</td>
<td>0.155</td>
<td>2.586</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15X</td>
<td>16.6X</td>
<td>1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trackers</td>
<td>27.7M</td>
<td>140.6M</td>
<td>13.160</td>
<td>3.052</td>
<td>20.693</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.6X</td>
<td>6.8X</td>
<td>1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wikipedia – link – de</td>
<td>3.2M</td>
<td>65.8M</td>
<td>1.987</td>
<td>0.764</td>
<td>3.954</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2X</td>
<td>5.1X</td>
<td>1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HDO (k-core decomp) results

- ParK: k-core algorithm from IEEE Big Data 2014
- HDO run on NVIDIA P100 with Hornet data structure.

| Name               | |V| | |E| | HDO (sec.) | ParK (sec.) | igraph (sec.) |
|--------------------|--------|------|--------|-----------------|-----------------|-----------------|
| dblp – author      | 5.5M   | 8.6M | 0.635  | 129.2X          | 1.595           | 82.066          |
| patentcite         | 3.8M   | 16.5M| 5.200  | 63.8X           | 13.294          | 331.538         |
| soc – LiveJournal1| 4.8M   | 42.9M| 60.755 | 25.9X           | 487.112         | 1572.985        |
| soc – pokec – relationships | 1.6M | 22.3M| 2.756  | 85.9X           | 6.488           | 235.790         |
| trackers           | 27.7M  | 140.6M| 1006.954| 4.7X           | 1148.638        | 4725.317        |
| wikipedia – link – de | 3.2M | 65.8M| 266.923| 11.3X           | 1397.323        | 3003.166        |