G-Store: High Performance Graph Store for Trillion-edge Processing

Pradeep Kumar and H. Howie Huang
The George Washington University
Introduction

- Graph is everywhere

- Applications include
  - Social Networks
  - Computer Networks
  - Protein Structures
  - Recommendation Systems
  - And many more ...

- Graph size is reaching to Trillion edges
Motivation

Trillion-edge graph

= 

Lower capacity but Fast access +

Higher capacity but Slow access
Outline

01 - Observations

02 – G-Store Architecture

03 - Experiment

04 – Conclusion and QA
Observation 1: Graph Size

- PageRank algorithm on Kron-28-16 graph
  - $2^{28}$ vertices
  - $2^{33}$ edges

- Graph sizes vary
  - 128GB if using 16-byte for an edge-tuple
  - 64GB for 8-byte edge-tuple

- Smaller graph size leads to better performance
Observation 2: Metadata Access Localization

- PageRank on 2-D partitioned graph
- In-memory processing performance
- Partition count affects metadata locality
Observation 3: Streaming Memory Size

- Small streaming memory does not significantly affect IO performance
- Use rest of the memory for caching to improve algorithm performance
G-Store

01 Graph Size
- Symmetry
- SNB Format

02 Metadata
- Grouping
- On-Disk Layout

03 Memory
- Proactive Caching
- Slide-Cache-Rewind

Graph Algorithm

G-Store
- Tile-based Representation
- On-Disk Grouping And Tile-based I/O
- Memory Management

Storage Devices
Tile-based Representation

(a) Example graph

(b) Edge-list

(c) 2-D Partitioned Graph

(d) Symmetry advantage

(e) Smallest Number of Bits (SNB) Format
Tile Advantages

- G-Store achieves upto 8x space saving
  - 2x due to Symmetry
  - Upto 4x due to Smallest Number of Bit representation (SNB)

- Processing can run directly on top of the SNB format with no need for conversion
  - Use a new relative pointer per tile for each algorithmic metadata
  - Please refer to the paper for details
Small Algorithm Change Needed for Tile

Algorithm 1 BFS on the partition[i,j] of undirected graph

1: \textit{edge} \leftarrow \textit{get\_edge\_ptr}(i,j);
2: \textbf{for} \ k \leftarrow 1, \textit{edge\_count}(i,j) \textbf{do}
3: \hspace{1em} \textit{src} \leftarrow \textit{edge}[k].\textit{src};
4: \hspace{1em} \textit{dst} \leftarrow \textit{edge}[k].\textit{dst};
5: \hspace{2em} \textbf{if} \ \textit{depth}[\textit{src}] == \textit{level} \ & \ \textit{depth}[\textit{dst}] == \textit{INF} \textbf{then}
6: \hspace{3em} \textit{depth}[\textit{dst}] \leftarrow \textit{level} + 1;
7: \hspace{2em} \textbf{end if}
8: \hspace{1em} \textbf{// Added code for new storage format}
9: \hspace{2em} \textbf{if} \ \textit{depth}[\textit{dst}] == \textit{level} \ & \ \textit{depth}[\textit{src}] == \textit{INF} \textbf{then}
10: \hspace{3em} \textit{depth}[\textit{src}] \leftarrow \textit{level} + 1;
11: \hspace{2em} \textbf{end if}
12: \textbf{end for}
Tile Properties

- Tile size of Twitter graph
  - Power law distribution

- Tile metadata size
  - All smaller than LLC

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Metadata Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page-Rank</td>
<td>256KB</td>
</tr>
<tr>
<td>Connected Components</td>
<td>256KB</td>
</tr>
<tr>
<td>BFS</td>
<td>64KB</td>
</tr>
</tbody>
</table>
Grouping and On-disk Layout

- Combine $q^2$ tiles into a physical group

- Layout of tiles in disk
  - Reading tiles sequentially provides better hit ratios on LLC

- Tile is a basic unit for Data Fetching, Processing and Caching

- Use Linux AIO
Grouping and On-disk Layout: Advantage

- 256*256 grouping has fewest LLC operations (loads, store) and misses

- Improves performance by 57%
Proactive Caching

- Divide memory into streaming and caching

- Rule 1: at the end of the processing of row\[i\]
  - We know whether row\[i\] would be processed in the next iteration
  - Hints can be used later

<table>
<thead>
<tr>
<th>Tile[0, 0]</th>
<th>Tile[0, 1]</th>
<th>Tile[1, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 1), (0, 3), (1, 2)</td>
<td>(0, 4), (1, 4), (2, 4)</td>
<td>(4, 5), (5, 6), (5, 7)</td>
</tr>
</tbody>
</table>

- Row[0]  
- Row[1]

- Rule 2: if row\[i\] is not needed for next iteration
  - Then we know that Tile\[i,i\] will not be needed
  - We only have partial information about other individual tiles
Slide-cache-rewind

Slide

\[ T_0 \quad \text{IO} \quad \text{Free Memory} \]

\[ T_1 \quad \text{Processing} \quad \text{IO} \quad \text{Free Memory} \]

\[ T_3 \quad \text{Cache pool} \quad \text{Processing} \quad \text{IO} \quad \text{Free Memory} \]

\[ T_i \quad \text{Cache pool} \quad \text{Processing} \quad \text{IO} \]
Slide-cache-rewind

- Proactive Caching
  - Less IO due to reuse
  - Provide hints for proactive caching policy
  - Evict unwanted data from cache pool
Experimental Setup

- Dual Intel Xeon CPU E5-2683, 14 core each, hyper-threading enabled = 56 threads
- 8GB streaming and caching memory
- Cache
  - 32KB data and instruction L1 cache
  - 256KB L2 cache
  - 16MB L3 cache
- LSI SAS9300-8i HBA, 8 Samsung EVO 850 512GB SSD
- Linux Software RAID0 with 64KB stripe size
## Graph Datasets & Space Saving

<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Type</th>
<th>X-Stream Size</th>
<th>FlashGraph Size</th>
<th>G-Store Size</th>
<th>Space Saving Over X-Stream</th>
<th>Space Saving Over FlashGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kron31-256</td>
<td>Undirected</td>
<td>8TB</td>
<td>4TB</td>
<td>2TB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Kron-33-16</td>
<td>Undirected</td>
<td>4TB</td>
<td>2TB</td>
<td>512GB</td>
<td>8x</td>
<td>4x</td>
</tr>
<tr>
<td>Kron-30-16</td>
<td>Undirected</td>
<td>256GB</td>
<td>128GB</td>
<td>64GB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Kron-28-16</td>
<td>Undirected</td>
<td>64GB</td>
<td>32GB</td>
<td>16GB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Rmat-28-16</td>
<td>Undirected</td>
<td>64GB</td>
<td>32GB</td>
<td>16GB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Random-27-32</td>
<td>Undirected</td>
<td>64GB</td>
<td>32GB</td>
<td>16GB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Twitter</td>
<td>(Un-)Directed</td>
<td>14.6GB</td>
<td>14.6GB</td>
<td>7.3GB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Friendster</td>
<td>(Un-)Directed</td>
<td>19.26GB</td>
<td>19.26GB</td>
<td>9.63GB</td>
<td>4x</td>
<td>2x</td>
</tr>
<tr>
<td>Subdomain</td>
<td>(Un-)Directed</td>
<td>15.22GB</td>
<td>15.22GB</td>
<td>7.6GB</td>
<td>4x</td>
<td>2x</td>
</tr>
</tbody>
</table>
Performance on Trillion-edge Graph

- Kron-31-256: $2^{31}$ vertices with $2^{40}$ edges
- Kron-33-16: $2^{33}$ vertices with $2^{38}$ edges

<table>
<thead>
<tr>
<th>Graph</th>
<th>BFS</th>
<th>Page-rank</th>
<th>WCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kron-31-256</td>
<td>2548.54</td>
<td>4214.54</td>
<td>1925.13</td>
</tr>
<tr>
<td>Kron-33-16</td>
<td>1509.13</td>
<td>1882.88</td>
<td>849.05</td>
</tr>
</tbody>
</table>

- One iteration of Pagerank:
  - G-Store: **14 min** in a **single machine** for a trillion-edge graph
  - Ching et al*: **3 min** using **200 machines** for similar-sized graph

*Ching et al. One Trillion Edges: Graph Processing at Facebook-scale. Proceedings of the 41st International Conference on Very Large Data Bases (VLDB15)*
### Performance Comparison

- **Over X-Stream**
  - 17x(BFS), 21x (PageRank), 32x(CC/WCC) speedup for Kron-28-16
  - Others are similar. See paper for details.

- **Over FlashGraph**
  - 1.5x (CC/WCC), 2x (PageRank), 1.4x (BFS, undirected)
  - Slightly poor for BFS on directed graph due to no space saving in smaller graph

![Graph showing speedup comparison between FlashGraph and G-Store](image-url)
Scalability on SSDs

- RAID0: 64K stripe size
- Close to 4x speedup for 4-SSDs
- Upto 6x speedup for 8-SSDs
  - PageRank becomes compute intensive at 8-SSD configuration
Slide-Cache-Rewind

- Base Policy (No cache): 4GB segment size (two)
- Cache+Rewind Policy: 7.5GB cache, 256MB segments (two)
- 60% (BFS), 35% (PageRank) and 35% (WCC)
Cache Size

- For Kron-28-16 graph from 1GB to 8GB
  - Average 30% speedup

- For Twitter graph from 1GB to 4GB
  - Average 41% speedup
Conclusion

- **SNB**: Space efficient representation
- **Grouping**: Optimal utilization of hardware cache
- **Slide**: Complete overlapping of IO and compute
- **Cache**: Graph specific proactive caching policy
- **Rewind**: Using the last drop of memory
Thank You

- Email: pradeepk@gwu.edu and howie@gwu.edu

- Graph software repository
  - [https://github.com/iHeartGraph/](https://github.com/iHeartGraph/)
  - G-Store: High-Performance Graph Store for Trillion-Edge Processing (SC’16)
  - Enterprise: Breadth-First Graph Traversal on GPUs (SC’15)
  - iBFS: Concurrent Breadth-First Search on GPUs (SIGMOD’16)