Fast Parallel Subgraph Matching on the GPU

Leyuan Wang, Yangzihao Wang, John D. Owens
University of California, Davis, CA, USA

Objectives

Graphs have been used to model interactions between entities in a broad spectrum of applications. Subgraph patterns help to understand the underlying structure of these graphs. We address the problem of Subgraph Matching (SM), finding all embeddings of a query graph \( q \) in a large data graph \( G \). SM can be easily modified to handle graph homomorphism which is the RDF pattern matching semantics, by just removing the injectivity constraint. RDF is primarily designed for building the Semantic web and has a large part of leaf nodes. We note that joining order selection and reducing intermediate results are considerably more challenging. We hope that the most challenging ones will serve as interesting future work developing frameworks for those particular methodologies.

Parallel Subgraph Matching

Input: Query Graph \( Q \), Data Graph \( G \).
Output: Number of matched query graphs \( count \) and listings of all occurrences \( outputlist \).

1. procedure initialize_candidate_set\((Q, G)\)
2. \( advance(G) \) \& fill \( c\_set \) based on node label and degree
3. end procedure
4. \( c\_set \) \( \rightarrow \) candidate set
5. procedure collect_candidate_edges\((G, Q, c\_set)\)
6. \( advance(G)\) \& label candidate edges with query_edge_id
7. \( filter(G)\) \& collect candidate edges
8. return \( ELIST \)
9. end procedure
10. parallel for each candidate edge combination \( \{e_1, e_2, \ldots, e_n\} \)
11. if \( \{e_1, e_2, \ldots, e_n\} \) satisfy intersection rules
12. Write \( \{e_1, e_2, \ldots, e_n\} \) to \( outputlist \)
13. Add 1 to \( count \)
14. end if
15. return \( count, outputlist \)
16. end procedure

Conclusion

In this work, we introduce an efficient method which takes advantage of GPU parallelism to deal with large-scale subgraph matching problem. We achieve up to 3.5× speedup against previous state-of-the-art method on the GPU. We also achieve significant speedups compared with specific triangle counting algorithm implementations. Our method, despite its memory bound, shows good performance on mesh like graphs that contain a large part of leaf nodes.

Dataset Description

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
<th>Max Degree</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>coAuthorsCiteseer</td>
<td>227,320</td>
<td>1,628,268</td>
<td>1372</td>
<td>rs</td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td>549,486</td>
<td>30,491,458</td>
<td>3299</td>
<td>rs</td>
</tr>
<tr>
<td>road_central</td>
<td>14,081,816</td>
<td>33,866,826</td>
<td>8</td>
<td>rm</td>
</tr>
<tr>
<td>kron_n17</td>
<td>131,072</td>
<td>10,227,970</td>
<td>28804</td>
<td>rm</td>
</tr>
<tr>
<td>delamay_n20</td>
<td>1,048,576</td>
<td>6,291,372</td>
<td>17</td>
<td>rm</td>
</tr>
<tr>
<td>enron</td>
<td>69244</td>
<td>549216</td>
<td>1391</td>
<td>rs</td>
</tr>
<tr>
<td>gomalia</td>
<td>196578</td>
<td>1900654</td>
<td>394</td>
<td>rs</td>
</tr>
</tbody>
</table>

Table: Dataset Description Table. The edge number shown is the number of directed edges when the graphs are treated as undirected graphs and de-duplicate the redundant edges. Graph types are: r: real-world, s: scale-free, and m: mesh-like.

Experiments & Results

![Figure: Speedups on Gowalla (top) and Enron (bottom) datasets of both our PSM and a state-of-the-art GPU implementation by Tran et al. [1].](image)

![Figure: Number of Query vertices vs. Running Time (msc).](image)

References


In Proceedings of the 1st High Performance Graph Processing Workshop, HPGP ’16, May 2016.


Contact Information

Email: leywang@ucdavis.edu

Challenges

- GPU operations are based on warps (which are groups of threads to be executed in single-instruction-multiple-data fashion), so different execution paths generated by backtracking algorithms may cause a warp divergence problem.
- Irregular memory access patterns cause memory accesses to become uncoalesced, which degrades GPU performance.

Our Solution

Our method follows a filtering-and-joining procedure and is implemented using the Gunrock programming model [3].

Filtering Phase. Prime away candidate vertices that cannot contribute to the final solutions using the massively parallel advance and filter operations in Gunrock.

Joining Phase. We do the actual joining only when the candidate edges satisfy the intersection rules in order to reduce the number of intermediate results and consequently, the amount of required memory.

Gunrock Programming Model

- Advance: generates a new frontier (in green) via visiting the neighbor vertices/edges in the current frontier (in blue).
- Filter: removes elements from a frontier via validation test.
- Compute: user-defined vertex/edge-centric computations that run in parallel, can be combined with advance or filter.