A GPU-based Graph Pattern Mining System

Lin Hu
Peking University
Beijing, China
hulin@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

ABSTRACT

Graph pattern mining (GPM) is getting increasingly important in recent years. Many graph pattern mining frameworks try to use universal primitives to deal with various graph pattern mining tasks. However, most of them suffer from unsatisfactory performance because of the exponential complexity of GPM. GPU is a new hardware with great parallelism, and many graph algorithms have achieved significant performance improvements on GPU.

In this demo, we propose a graph pattern mining framework on GPU, called GAMMA. GAMMA proposes effective and flexible interfaces for users to implement their mining tasks conveniently. GPM has extensive intermediate results in parallel environments. We make full use of host memory to deal with large-scale graphs and extensive intermediate results. We also present several optimizations to process large graphs. GAMMA has great scalability and performance advantages compared with state-of-the-art graph mining works in experiments.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

graph pattern mining; GPU; large graphs

ACM Reference Format:
Lin Hu and Lei Zou. 2022. A GPU-based Graph Pattern Mining System. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3511808.3557192

1 INTRODUCTION

Graph pattern mining (GPM) is getting increasingly important recently. Given an input graph, graph pattern mining aims to find subgraphs of interest. It includes many graph algorithms such as subgraph matching (SM), frequent pattern mining (FPM), motif counting, k-clique counting (kCL) and triangle counting. Figure 1 gives examples of two graph pattern mining tasks, and we will present how to implement them in our framework. So far, many graph pattern mining frameworks aim to provide universal solutions for those GPM algorithms: users can use the primitives provided by those frameworks to build their mining algorithms conveniently. However, most of those frameworks suffer from unsatisfactory performance because of the exponential search space of GPM. For example, a state-of-the-art GPM framework, Arabesque [5], spends 1.65 hours on finding all length-3 frequent patterns in a graph with only one million edges.

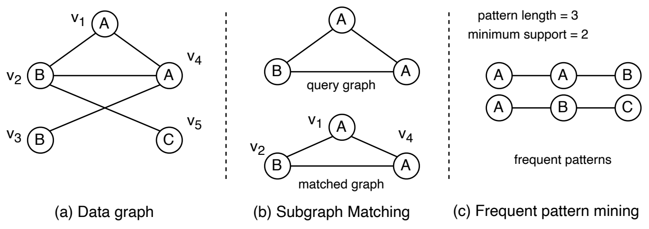


Figure 1: Two specific graph mining tasks.

One way to improve the performance of GPM is to employ hardware assist. Due to the massive parallelism of GPU, the performance of graph algorithms can be improved significantly. However, GPM algorithms usually produce massive intermediate results; on the other hand, GPU device memory is limited compared with host memory. For example, Pangolin [2], the only existing graph pattern mining framework on GPU, cannot perform FPM on a graph with only 17M edges because of the limit of device memory capacity.

Table 1: Average performance improvement of GAMMA compared with other state-of-the-art graph mining works.

	Pangolin[2]	Peregrine[4]	GraphMiner[1, 3]	GSI[6]
speedup	65.5%	79.5%	66.9%	56.7%

Targeting the above two problems, i.e., performance and scalability, we develop a graph pattern mining framework for large graphs (*GAMMA* for short) on an out-of-core GPU system. To the best of our knowledge, our GAMMA is the first GPM framework on GPU that uses both device memory and host memory to accommodate large graphs and massive intermediate results. GAMMA has orders of magnitude better scalability in graph size than Pangolin, the only existing GPM framework on GPU. GAMMA can process graphs with about 500M edges, which uses about 300 GB of host memory in processing. We also aim at high performance in framework design, such as data structure design and primitive implementation. Using FPM and SM as examples, we achieve good performance improvement compared with state-of-the-art frameworks on GPU and multi-core CPU, even compared with some GPU-based specific graph algorithms such as GSI [6] in subgraph matching. The performance advantages are shown in Table 1.

Generally, GAMMA makes the following technique contributions:

(1) Self-adaptive host memory access methods. There are two kinds of implicit host memory access methods from GPU: unified memory and zero-copy memory. The former is more suitable for data with good spatial or temporal locality, and the latter is friendly to isolated and infrequently accessed data. We put graph data in host memory, and dynamically determine the access method for each page depending on the access locality of the pages in the running process. This helps to enlarge the graph size that our framework can process and smooth the gap between host memory and device memory.

(2) Primitive optimizations. We adopt "extension-aggregation-filtering" as the primitives for users to build various graph pattern mining algorithms. However, there are some new challenges as the graph size increases, especially when graphs are beyond the capability of device memory. Therefore, we propose three optimizations to our primitives to address those problems. Generally, we propose a dynamic memory allocation strategy to avoid writing conflict, pre-merge adjacency lists to avoid redundant computation, and reduce the workload of out-of-core sort in the aggregation primitive. Those optimizations play key roles in performance improvement.

We will demonstrate GAMMA to implement classical GPM algorithms to show the user-friendly interfaces, low-code programming model and high efficiency of GPM implementations.

2 SYSTEM OVERVIEW

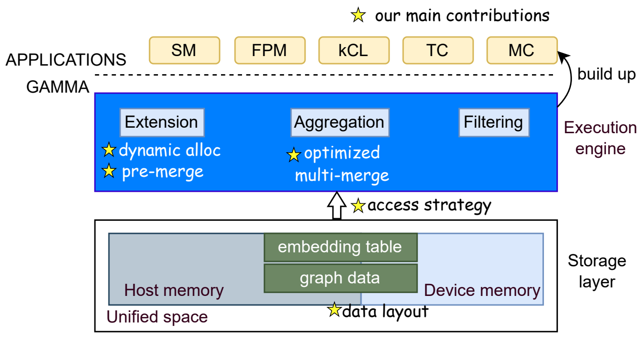


Figure 2: Architecture of GAMMA

Figure 2 depicts the architecture of GAMMA, including the storage layer and the execution engine. Generally, there are two essential parts in storage in GPM framework: graph data and intermediate results, and we refer to the latter as *embedding table* in our paper. In the storage layer, we assume that both graph data and the embedding table are resident in both GPU device memory and host memory. We propose a self-adaptive graph data storage and access strategy on a heterogeneous computing system involving CPU and GPU. In the out-of-core GPU execution engine, we use flexible and effective primitives, i.e., "extension-aggregation-filtering", to process the embedding table, and propose our optimized execution algorithms of the three primitives over the embedding table. Users can use these primitives to build various GPM algorithms conveniently.

Generally, we call the subgraphs of interest in GPM as *patterns*, and refer to those instances found in the input graph as *embeddings*.

Data structures visible to users

1. constraint c; //describing how to pruning embeddings
2. embedding_table ET; //the data structure of intermediate results
3. graph_data G_d; //the data graph
4. query_graph G_q; //the query graph

Interfaces visible to users

1. Vertex_Extension (embedding_table ET, graph_data G_d);
2. Edge_Extension (embedding_table ET, graph_data G_d);
3. Aggregation (embedding_table ET, map_function mf);
4. Filtering (embedding_table ET, constraint c);

Figure 3: Interfaces of GAMMA

Figure 3 shows all the interfaces and primitives visible to users. Here we introduce them briefly.

- The *embedding table ET* is the set of intermediate results, and the *data graph* G_d is the input graph represented as compressed sparse rows (CSR). GAMMA targets large graphs, therefore both data structures are resident at host memory. Users can specify the *constraints c*, such as the frequency threshold in FPM, and the *query graph* G_q in SM.
- The extension primitive takes the embedding table as input, and extends the length of embeddings in it by one. Embeddings can be edge-induced or vertex-induced; accordingly, there are two extension strategies (edge-extension and vertex-extension) to support flexible graph mining tasks.
- Aggregation primitive maps each embedding in the embedding table into a pattern, and aggregates those mapped patterns to obtain some statistical information.
- Filtering is a primitive following the "extension" and "aggregation" to prune embeddings that do not satisfy given constraints.

Those interfaces make it easy for users to build various graph mining tasks. Here we give an example of implementing subgraph matching, a vital GPM algorithm, with our framework GAMMA.

Algorithm 1: WOJ Subgraph Matching

```
Input: query graph $G_q$, data graph $G_d$.
Output: subgraph matching results.
1 Let $\delta_v$ denote the matching order of vertices in $G_q$;
2 $ET \leftarrow$ all matched vertices to the first vertex in $\delta_v$;
3 foreach unmatched vertex $v \in \delta_v$ do
4     Vertex_Extension($ET, G_d$) ;
5     Filtering($ET, Constraint = G_q$) ;
6 end
7 output_result($ET$) ;
```

Given a query graph G_q, the subgraph matching aims to find all subgraphs in the data graph G_d that are isomorphic to G_q. There are usually two methods for subgraph matching: binary join and worst-case optimal join (WOJ). The former extends one edge at a time, and the latter extends one vertex at a time. Here we demonstrate subgraph matching implementation using vertex-centric extension in Algorithm 1. The initial embeddings in SM are one-column embedding table, matching the first vertex in G_q. In each iteration, we process one query vertex. For each embedding in the embedding table, we consider all possible vertex extensions for a given query vertex (line 4). Extended embeddings can be safely filtered if violating the subgraph isomorphism of G_q (line 5). Finally, we output all embeddings as results. Binary join can be implemented using GAMMA with a similar process, except that it uses edge extension.

GAMMA frees users from tedious programming details, especially complicated primitive optimizations, graph partition, host memory access and large-scale intermediate results maintenance between device memory and host memory in the traditional specific out-of-core GPU algorithms. In GAMMA, users only need to focus on implementing the algorithm logic using the three primitives. GPU execution optimizations, memory access and data organization are all handled by GAMMA, which are transparent to users.

3 TECHNIQUES

3.1 Host Memory Access

To support large graph processing on GPU, we must use host memory since the device memory is limited (i.e., Tesla V100 has only 16 GB of device memory). Basically, there are two kinds of strategies to access host memory, i.e., explicit memory access and implicit memory access. The former organizes the required data in host and transfers them to the device explicitly before launching GPU kernels, while the latter launches data transfer from GPU at runtime and accesses the required data on-the-fly. The critical problem for explicit host memory access is that it is often a task-specific access method, which is unsuitable for our comprehensive GPM framework. Thus, GAMMA employs the implicit memory access approach.

There are two implicit host memory access modes: unified memory and zero-copy memory. The former transfers data at the size of a page (4 KB), and caches pages in device for later access; while the latter does not have a cache in device, and transfers data at the unit of a cache line (128 Byte). Unified memory is suitable for data with good spatial or temporal locality, and zero-copy memory is friendly to infrequently accessed or isolated data.

Continuous embeddings in the embedding table often have the same parent embeddings. Thus, we use tree structures to compress the embedding table. Access to a batch of embeddings is continuous and concentrated, in which case we use unified memory to access the embedding table.

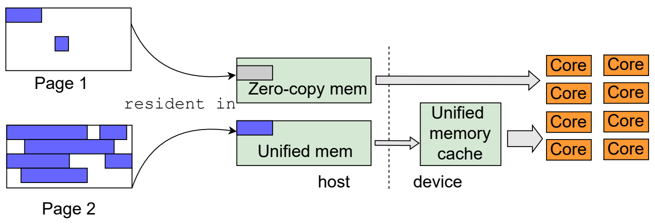


Figure 4: Two types of pages.

Access to graph data is more complicated. Some pages of graph data may be accessed multiple times in an extension, and there are some pages of which only a few cache lines are accessed. Figure 4 illustrates the two different cases, where accessed data in two pages are highlighted in blue. Apparently, they should be accessed in different manners. We design a self-adaptive access method for each page as follows. Firstly, we formally define spatial and temporal localities in Definitions 3.1 and 3.2, respectively. The former defines how much data in a page are accessed in one extension, and the latter shows how much data in a page are accessed in past extensions. We combined them together to obtain *Access Heat* (see Definition 3.3).

Definition 3.1 (Spatial Locality). The spatial locality of a page p in the i-th extension is defined by the access quantity of p, i.e.,

$$SpatialLoc_i(p) = \sum_{l(v)\in p \wedge l(v)\in A_i} |l(v)| \times times_i(l(v)) \tag{1}$$

where A_i denotes all accessed adjacency lists in the i-th extension, $l(v)$ denotes the adjacency list of vertex v, and $|l(v)|$ is its size. $times_i(l(v))$ denotes how many times $l(v)$ is accessed in the i-th extension.

Definition 3.2 (Temporal Locality). The temporal locality of a page p in the i-th extension is defined by the access quantity of p in the first i-1 extensions, i.e.,

$$TempLoc_i(p) = \sum_{j\le i-1} \sum_{l(v)\in p \wedge l(v)\in A_j} |l(v)| \times times_j(l(v)) \tag{2}$$

where A_j, $l(v)$, $|l(v)|$ and $times_j(l(v))$ have been introduced in defining spatial locality.

Definition 3.3 (Access Heat). The access heat of a page p is defined as follows:

$$AccHeat_i(p) = \frac{A_i}{\sum_{j\le i} A_j} \times SpatialLoc_i(p) + \frac{\sum_{j\le i-1} A_j}{\sum_{j\le i} A_j} \times TempLoc_i(p)$$

where A_j denotes the total accessed data in the j-th extension, $SpatialLoc_i(p)$ and $TempLoc_i(p)$ are defined in Equations 1 and 2.

The *Access Heat* combines spatial locality and temporal locality by the ratio of total accessed data, and is a good measurement of how likely a page is accessed in the procedure. Before each extension, we update the *AccHeat* of each page. Pages with high *AccHeat* will be accessed by unified memory, and other pages will be accessed by unified memory. In this way, different pages can be accessed through different access manners to make the most of different host memory access manners.

3.2 Primitive Optimizations

Processing large graphs brings about some new challenges, which become performance bottlenecks for GPM on GPU. Here we briefly introduce those challenges, and propose our solutions accordingly.

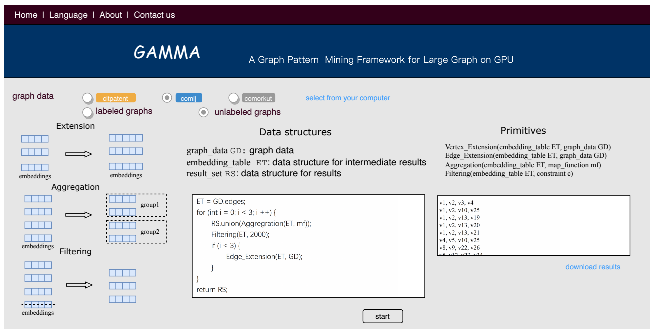
Dynamic memory allocation. Each thread extends one embedding, producing an uncertain number of new embeddings. Therefore, parallel threads do not know the positions they should start writing. Existing methods solve this with one extra extension or space pre-allocation, which brings a lot of additional time cost or space cost. We propose a dynamic memory allocation strategy: we divide the memory space into many memory blocks, and each running thread asks for a memory block to write new results; after a memory block is full, the thread will ask for another one. This dynamically adjusts the memory dispatched to each thread in run-time, and does not introduce extra space cost or time cost.

Grouping embeddings with common prefixes. Embeddings produced from the same parent share the same prefixes. Therefore, there is much redundant computation for them. For example, consider two instances (u_1, u_2, u_3, u_4) and (u_1, u_2, u_3, u_5) in 5-clique problem: when extending the fifth vertex in those two instances, they both need to intersect the adjacency lists of u_1, u_2 and u_3. Based on this observation, we first group embeddings according to their prefixes before extension, and assign embeddings in the same group to the same thread to save redundant computation.

Optimized external sort. The mapped patterns in the aggregation primitive often need to be sorted, and their size may exceed the capacity of device memory, in which case external sort on GPU is needed. Considering the external sort in CPU, we also use the "divide-sort-merge" process to solve this problem. In the merge stage, we firstly partition the original merging task into many small tasks to dispatch them to multiple threads. In each subtask, the element in each segment searches for matches in all other segments; we propose to replace half workloads with prefix-scan and vector addition, which improves the merging performance.

4 DEMONSTRATION

4.1 System Interface



graph_data GD: graph data
embedding_table ET: data structure for intermediate results
result_set RS: data structure for results

```
Vertex_Extension(embedding_table ET, graph_data GD)
Edge_Extension(embedding_table ET, graph_data GD)
Aggregation(embedding_table ET, map_function mf)
Filtering(embedding_table ET, constraint c)
```

```
ET = GD.edges;
for (int i = 0; i < 3; i ++) {
    RS.union(Aggregration(ET, mf));
    Filtering(ET, 2000);
    if (i < 3) {
        Edge_Extension(ET, GD);
    }
}
return RS;
```

```
v1, v2, v3, v4
v1, v2, v10, v25
v1, v2, v13, v19
v1, v2, v13, v20
v1, v2, v13, v21
v4, v5, v10, v25
v8, v9, v22, v26
```

Figure 5: Demonstration of GAMMA.

We demonstrate how to use GAMMA to build various graph pattern mining algorithms. Figure 5 gives an example of building frequent pattern mining with GAMMA. Users can upload their own datasets and specify whether the input graph is labeled.

GAMMA provides several user-visible data structures: graph data GD, data structure for intermediate results ET and results set RT. Users can use them and basic primitives to develop their mining algorithms. Figure 5 shows how to implement a frequent pattern mining algorithm with the pattern length as three and the minimum support as 2000 in GAMMA with a few lines of codes: firstly, we initialize the results set as all edges in the input graph; in each iteration, we aggregate all embeddings, filter our invalid embeddings, and extend all frequent patterns. Subgraph matching can also be implemented as the pseudocode in Algorithm 1. In fact, it is easy for users to build their own graph mining algorithms besides common GPM algorithms, such as only mining subgraph patterns with specific labels or discovering frequent parent patterns of a given subgraph. Generally, GAMMA frees users from tedious and complicated coding in the CPU/GPU cooperating computing platform and allows them to focus on GPM algorithm logic.

4.2 Performance

Table 2: All datasets in our experiments.

datasets	CP	CL	CO	EA	ER	CL×8	SL×5
nodes	6M	4M	3M	265K	37K	32M	24M
edges	17M	34M	117M	729K	368K	467M	481M
types	citation	social	social	email	email	synthetic	synthetic

We compare GAMMA with the state-of-the-art GPM frameworks, i.e., Peregrine [4] and Pangolin [2], and some specific GPM implementation, i.e., GraphMiner [1, 3] and GSI [6], to show the advantages of GAMMA in performance and graph scalability. We use both real-world and synthetic datasets, as shown in Table 2.

4.2.1 Subgraph matching. GSI is a subgraph matching implementation on GPU, and Peregrine is a state-of-the-art multi-core GPM framework on CPU. We compare the performance with them on subgraph matching, and the results are shown in Table 3. We do not compare with Pangolin and GraphMiner, since they do not have SM implementations. GAMMA performs much better than GSI and Peregrine on all large graph datasets except for two small datasets (EA and ER). For small datasets, the preparation of host memory usage in GAMMA accounts for a large portion of the total running time. GAMMA achieves 56.7% performance improvement over GSI; compared with Peregrine, GAMMA achieves 76.1% performance improvement.

Table 3: Performance of subgraph matching (sec).

	CP	CL	CO	EA	ER	CL×8	SL×5
GSI	1.12	2.01	2.84	0.35	0.21	8.3	7.2
Peregrine	1.46	4.47	11.3	0.24	0.1	11.7	12.4
GAMMA	0.76	0.86	1.39	0.39	0.30	1.84	2.5

4.2.2 Frequent pattern mining. Pangolin is the only existing GPM framework on GPU, but it only uses device memory for graph data and intermediate results. GraphMiner is a graph mining algorithm library on CPU, which combines several state-of-the-art graph mining frameworks. We compare the performance with those two works as well as Peregrine on FPM, as shown in Table 4. Pangolin only works on the two smallest datasets (EA and ER) and fails in all other large graphs, as denoted in "-" in Table 4. GAMMA has 65.5% performance improvement over it. GAMMA performs significantly better than GraphMiner and Peregrine on large and medium-size datasets, achieving an average of 82.9% and 66.9% performance improvement over Peregrine and GraphMiner, respectively.

Table 4: Performance of frequent pattern mining (sec).

	CP	CL	CO	EA	ER	CL×8	SC×5
Pangolin	-	-	-	12.1	7.2	-	-
GraphMiner	30.0	168	93.6	3.3	1.8	-	-
Peregrine	104	154	62.5	2.9	2.3	-	-
GAMMA	23.1	19.1	10.3	4.6	2.2	31.4	30.5

ACKNOWLEDGMENTS

This work was supported by NSFC under grant 61932001, U20A20174. The corresponding author of this paper is Lei Zou (zoulei@pku.edu.cn).

REFERENCES

[1] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*. 378–391.

[2] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[3] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. 2021. FlexMiner: a pattern-aware accelerator for graph pattern mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 581–594.

[4] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 13:1–13:16. https://doi.org/10.1145/3342195.3387548

[5] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[6] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. Gsi: Gpu-friendly subgraph isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1249–1260.