GAMMA: A Graph Pattern Mining Framework for Large Graphs on GPU

Lin Hu†, Lei Zou†, M. Tamer Özsu‡
†*Peking University, China;* ‡*University of Waterloo, Canada;*
†{hulin,zoulei}@pku.edu.cn, ‡tamer.ozsu@uwaterloo.ca

***Abstract*—Graph pattern mining (GPM) is getting increasingly important recently. There are many parallel frameworks for GPM, many of which suffer from performance. GPU is a powerful option for graph processing, which has excellent potential for performance improvement; however, parallel GPM algorithms produce a large number of intermediate results, limiting GPM implementations on GPU.**

In this paper, we present GAMMA, an out-of-core GPM framework on GPU, and it makes full use of host memory to process large graphs. Specifically, GAMMA adopts a self-adaptive implicit host memory access manner to achieve high bandwidth, which is transparent to users. GAMMA provides flexible and effective interfaces for users to build their algorithms. We also propose several optimizations over primitives provided by GAMMA in the out-of-core GPU system. Experimental results show that GAMMA has scalability advantages in graph size over the state-of-the-art by an order of magnitude, and is also faster than existing GPM systems.

***Index Terms*—graph pattern mining, large graphs, GPU**

I. Introduction

The importance of graph algorithms in many fields is well-recognized: chemical engineering [1], social network [2], [3] and financial market [4]. Significant attention has been paid to graph pattern mining (GPM) tasks that discover graph patterns satisfying some criteria [5]–[9]. This class of workloads involves subgraph matching (SM) [10], frequent pattern mining (FPM) [11], [12] and k-clique (kCL) computation [13]. GPM algorithms usually produce a large number of intermediate results, making them more challenging. For example, exploring length-4 embeddings over cit-Patent (a dataset with 16.5 M edges) produces 13.5 billion intermediate results [7]. In this paper, we focus on efficient computation of GPM algorithms using hardware accelerators.

There are two different roadmaps to develop GPM algorithms. One is to design an efficient graph algorithm for a *specific* task, such as subgraph matching [10], [14], [15] and FPM [11], [12]. The second is to define a generic *framework*, which incorporates efficient primitives that can be used to specify GPM algorithms. These primitives are tuned to address the computational commonalities among GPM tasks such as their computationally heavy nature, their tendency to perform random access to data, and the production of massive intermediate results by many well-known GPM algorithms. In this paper we follow the second approach and develop GAMMA (graph pattern mining framework for large graphs), a framework that incorporates primitives that can be efficiently executed on GPUs and can be used to implement GPM algorithms such as FPM and SM.

Many GPM frameworks have been proposed [5], [8], [9], [16], most of which are CPU-based. They generally have unsatisfactory performance due to the exponential search space of GPM and limitations of CPU-only computation. For example, Arabesque [5], a state-of-the-art GPM framework, spends 1.65 hours to find all length-3 frequent patterns in a graph with one million edges.

GPM is a class of algorithms that can benefit from hardware assistance, specifically GPU processing. GPU provides massive parallelism with a large memory bandwidth compared to a CPU, making it suitable for graph pattern mining. Most existing GPU-based works focus on designing specific GPM algorithms [10], [13], [17] rather than a comprehensive framework. To the best of our knowledge, Pangolin [8] is the only GPU-based GPM framework. It works on the assumption that graphs and intermediate results can be resident in GPU device memory. However, as noted earlier, GPM algorithms often produce extensive intermediate results, and GPU device memory is quite limited (e.g., 16 GB for Tesla V100). Thus, Pangolin cannot deal with large graphs.

To deal with large graphs on GPU, existing works [17]–[20] partition graphs and explicitly transfer them to GPU to process them. However, they require task-specific partitioning strategies, and this approach causes redundant memory transfer and extra data reorganization cost. It is desirable to avoid these overheads.

The goal of this paper is to design an out-of-core GPM framework for graphs that are too large to fit GPU device memory. We have two main challenges: one is how to *store and access* large graph data and intermediate results on CPU-GPU heterogeneous platform; the other is to address the *computational bottlenecks* due to processing large graphs on out-of-core GPU systems.

To address the first issue, we adopt the *implicit* host memory access approach, where host memory and device memory become a unified address space. This style of access assures that the data required by the device can be fetched from CPU at run-time. There are two kinds of implicit memory access modes with different characteristics: unified memory and zero-copy memory. Accessing unified memory may cause additional data migration, but it has buffers in the device; zero-copy memory has no buffer in the device, and has little migration cost. Thus, unified memory is friendly to data with good temporal and spatial locality, while zero-copy memory is suitable for isolated and infrequently accessed data. Neither works particularly well for GPM because of the diversity of access patterns to graphs. In this paper, we propose a self-adaptive access approach based on a quantitative model of the access. We also design data structures considering data locality to smooth the bandwidth gap between host memory and device memory.

The second challenge is that the computational complexity of GPM algorithms increase quickly as the graph size grows, and the performance issues of in-core GPU systems become even more serious in out-of-core GPU platforms (see Section V-B). These include the uncertain amount of output produced by threads and the large amounts of computational redundancy, and sorting data whose size exceeds device memory is a new challenge. We develop three optimizations to the primitives to address these issues: (1) design a dynamic device memory allocation strategy to address the uncertainty in the *extension* primitive, (2) group multiple extension processes to reduce redundant computation, and (3) implement an efficient sort method when the key size exceeds device memory.

TABLE I
CATEGORIES OF DIFFERENT GRAPH MINING WORKS.

	GPU		CPU
	in device	out of device	
frameworks	Pangolin [8]	GAMMA	Peregrine [16], Kaleido [7], Arabesque [5], Fractal [9]
specific algos	GSI [10], Tricore [17]	Guo et al. [15]	Sun et al. [21]

Our self-adaptive memory access approach enables GPU to process much larger graphs than what is currently possible; the proposed optimizations to primitives guarantee better performance and better scalability. These are our main contributions, and they are incorporated into GAMMA. To the best of our knowledge, GAMMA is the first out-of-core GPM framework to deal with large graphs that are beyond the capacity of device memory (see Table I). Programming GPM algorithms within GAMMA frees users from massive programming details, including complicated host memory access, maintaining large-scale intermediate results and primitive optimizations. We demonstrate this by building three GPM algorithms.

Experimental results show that GAMMA can support billion-scale graphs and has an order of magnitude better scalability in graph size than other GPM frameworks on GPU.

To summarize, we make the following contributions:

- We propose a GPM framework on GPU, called GAMMA, which uses host memory to deal with large graphs. It provides flexible and effective interfaces for users to build their algorithms.
- We build a self-adaptive method to determine when to use alternative modes of accessing host memory (unified and zero-copy), each of which is suitable for different situations. This helps to smooth the bandwidth gap between host memory and device memory.
- We propose three optimizations to existing GPM framework primitives based on the GPU architecture and graph mining tasks. These optimizations target large graphs.
- We conduct extensive experiments. The results show that GAMMA has great improvements in scalability and performance compared with state-of-the-art works.

II. PRELIMINARIES

A. Heterogeneous System Architecture

The architecture of a heterogeneous computing system involving CPU and GPU is shown in Fig. 1.

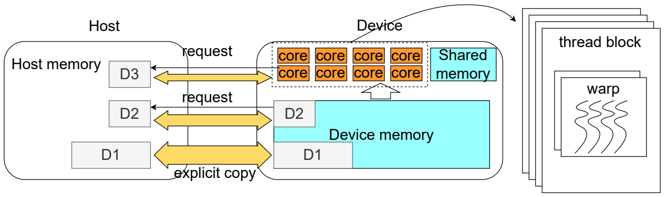


Fig. 1. GPU architecture.

Software. A *warp* is a group of threads, and threads in it run in Single Instruction Multiple Threads (SIMT) manner. Therefore, synchronization in a warp does not introduce extra cost. A *thread block* consists of several warps, and it is the largest unit for thread communication. Thread block synchronization has a much higher overhead than warp synchronization.

Hardware. GPU has thousands of cores, and they share device memory. Shared memory is on-chip memory managed by thread blocks. It is limited in size (about 48 KB per thread block) but has low access latency. Device memory is connected to host memory via PCIe. Data transfer between the host (CPU) and GPU is a critical part of GPU-optimized algorithms, and we discuss it in detail in the following subsection.

B. Host Memory Access

It is traditional in processing large graphs to use *explicit* memory transfer to move each portion of the graph to the device memory ($D1$ in Fig. 1). This can be achieved in two ways. The first approach [15], [17]–[19] is partitioning the large graph such that each partition fits into device memory, and partitions are iteratively loaded to device memory and processed. This method introduces extra data transfer cost and reduces the utilization of GPU. Furthermore, this task-specific data partitioning solution cannot support a universal GPM framework. The second solution is a fine-grained data transmission method proposed by Subway [20]. It collects the required data, reorganizes them into a compressed structure in the host, and transfers them to the device. Obviously, data extraction and reorganization on CPU are costly. Therefore, explicit memory transfer cannot be applied to large-scale GPM on an out-of-core GPU platform.

Implicit memory access unifies host memory and device memory into the same address space, and the required data can be fetched from CPU on-the-fly. This method is transparent to users and more suitable for general-purpose tasks. Furthermore, it overlaps data transfer and computation because threads issuing memory requests will be switched off GPU until the required data are fetched. Therefore, we use them for host memory access in GAMMA.

There are two implicit memory access modes: *unified memory* and *zero-copy memory*. Unified memory treats host and device memory as unified memory space, and data are resident in either side. When a memory access request (even a single byte) is issued from device to data resident in host, a page fault occurs, and a data page (typically 4 KB) is migrated from host to device and buffered. This leads to page-fault handling and long migration latency, but subsequent accesses to the same page can directly refer to the device buffer for the required data. $D2$ in Fig. 1 uses unified memory transfer.

Data access to zero-copy memory will cause data transfer at units of 128 bytes. Thus, it has almost no extra data migration cost. It does not have any buffer on the device. As a result, every time the device issues a memory access request to zero-copy memory, the required data will be transferred to the device. $D3$ in Fig. 1 uses zero-copy memory access.

In summary, unified memory is friendly to data with good spatial or temporal locality, in which case multiple accesses to the buffered data will make up for the time of page fault and long migration latency; zero-copy memory is suitable for isolated and infrequently accessed data because small data migration size assures low latency. One of our significant contributions is to design a self-adaptive strategy to determine proper host memory access manner for different pages.

III. GAMMA Design Overview

GPM involves finding subgraphs of interest in an input data graph G_d. In this paper, we refer to a subgraph to be found as a *pattern*, and each specific instance found in the data graph as an *embedding* or *instance*. Although GAMMA applies to all GPM tasks (e.g., triangle counting, motif counting, kCL, SM and FPM), in this paper we use subgraph isomorphism (SM) and frequent pattern mining (FPM) as running examples for illustration. These tasks are defined as follows:

- *Subgraph isomorphism.* SM finds in a data graph G_d all subgraphs (instances) isomorphic to a pattern P that can be represented as a query graph G_q.
- *Frequent pattern mining.* FPM finds all patterns P whose support (denoted as "sup") is at least a given threshold. P's support is the frequency of the instances of P in G_d.

A. Embedding Table

The collection of embeddings for a given pattern is organized as an *embedding table*. The embeddings can be organized in either vertex-oriented or edge-oriented fashion. Consequently, the embedding table can be vertex-oriented (called *v-ET*) or edge-oriented (called *e-ET*).

Fig. 2 shows examples of SM and FPM. The labels besides vertices (i.e., v_i and u_j) are vertex IDs that we introduce to simplify the description of the graph; similarly for edge IDs e_i and E_j. The labels (such as 'A') inside vertices are the actual vertex labels.In a v-ET T_v, each column corresponds to one vertex in the pattern. For example, vertex embedding (u_1,u_2,u_3) in the v-ET of Fig. 2(b) corresponds to pattern (v_1, v_2, v_3) in G_q. For e-ET T_e, each column corresponds to one edge in the pattern: the first column in e-ET of Fig. 2(b) records the matched edges of E_1 in G_q.

B. Execution Workflow

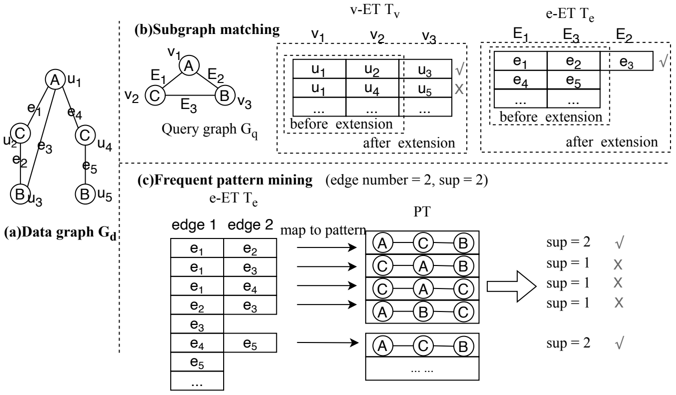


Fig. 2. An example of edge-extension and vertex-extension in SM and FPM.

GAMMA adopts a three-phase execution process: "extension-aggregation-filtering" [7]–[9].

1) Extension: The extension step takes an embedding table as input, and extends the length of each embedding in it by one. Depending on the type of embedding table that is used, two types of extensions are possible: *vertex-extension* and *edge-extension*. Each adds one possible vertex (or edge) to the existing embeddings.

Definition 3.1 (Extension): Given an embedding M, the *vertex extension* ($Ext_v(M)$) and *edge extension* ($Ext_e(M)$) of M are defined as follows:

$$\begin{aligned} Ext_v(M) = \{M \oplus u | u \in N_v(M)\} \\ Ext_e(M) = \{M \oplus e | e \in A_e(M)\} \end{aligned} \tag{1}$$

where $\oplus$ denotes adding one vertex or edge into the current embedding; $N_v(M)$ and $A_e(M)$ denote all neighbor vertices and adjacent edges to the instance M, defined as

$$\begin{aligned} N_v(M) = \textstyle\bigcup_{u' \in V(M)} N_v(u') - V(M) \\ A_e(M) = \textstyle\bigcup_{u' \in V(M)} A_e(u') - E(M) \end{aligned} \tag{2}$$

where $N_v(u')$ and $A_e(u')$ denote all neighbor vertices and adjacent edges to vertex u', $V(M)$ and $E(M)$ denote all vertices and edges in instance M.

Generally, FPM algorithm uses edge-extension, as shown in Fig. 2(c). SM can use both types of extension: edge extension can implement a *binary join* (query-edge-at-a-time) [22] and vertex extension can implement a *worst-case optimal join* (query-vertex-at-a-time) [23]. Fig. 2(b) includes examples of vertex-extension and edge-extension in SM. In v-ET T_v, initially there are two embeddings (u_1, u_2) and (u_1, u_4) that match (v_1, v_2) in G_q. They can be extended to (u_1, u_2, u_3) and (u_1, u_4, u_5), respectively. An analogous process exists for edge extension in e-ET T_e. These two different extension methods make our approach more flexible and effective in building various GPM algorithms.

2) Aggregation: This step maps an embedding table ET into a pattern table PT and computes an aggregation function over PT. Specifically, each embedding in ET is mapped to a pattern graph. For example, both embeddings (e_1, e_2) and (e_4, e_5) are mapped to the same pattern graph $(A\text{-}C\text{-}B)$ in FPM of Fig. 2(c). This can be achieved by computing graph canonical label [24][1]. Finally, the mapped patterns are aggregated over PT. For example, only pattern $(A\text{-}C\text{-}B)$ has support 2, since it has two instances.

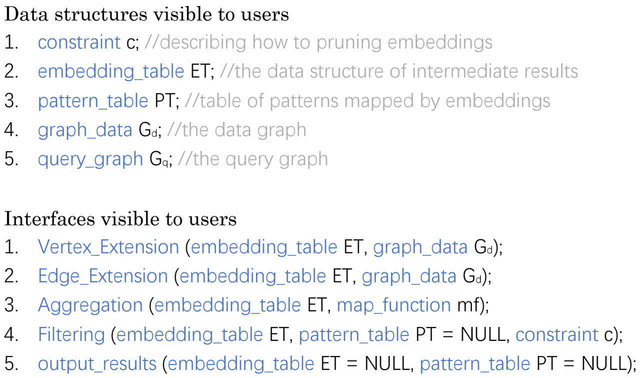
3) Filtering: GAMMA allows users to specify constraints on the embedding. For example, the extended embeddings should satisfy the given query graph's structure in SM; the support of the mined graph pattern should be no less than a given threshold in FPM. These conditions can be enforced through *filtering* following extension or aggregation.

Some of the pruning can be performed earlier even though the filtering step follows extension and aggregation. For example, in SM using vertex extension, extended embeddings violating the query graph's constraint can be pruned immediately. When extending the embedding (u_1, u_2) in Fig. 2(b), we only consider the common neighbors of u_1 and u_2.

Note that not every primitive is used in every specific algorithm, but the primitives are able to support various GPM algorithms.

C. Implementing GPM Tasks – Examples

We illustrate the application of the described workflow by implementing SM and FPM. Fig. 3 lists the data structures and interfaces visible to users in GAMMA.



Data structures visible to users

1. constraint c; //describing how to pruning embeddings
2. embedding_table ET; //the data structure of intermediate results
3. pattern_table PT; //table of patterns mapped by embeddings
4. graph_data G_d; //the data graph
5. 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, pattern_table PT = NULL, constraint c);
5. output_results (embedding_table ET = NULL, pattern_table PT = NULL);

Fig. 3. Data structures and interfaces visible to users in GAMMA.

1) Subgraph Isomorphism: This algorithm can be implemented using either binary join or worst-case optimal join (WOJ). We illustrate the latter using primitives in GAMMA.

We demonstrate WOJ implementation using vertex-centric extension in Algorithm 1. The initial embeddings in WOJ are one-column embedding table, matching the first vertex in G_q. In each iteration, we process one query vertex. For example, assume that we have all matched vertices corresponding to v_1 in Fig. 2(b) (line 2), and the next query vertex is v_2. For each embedding in T_v, we consider all possible vertex extensions (e.g., u_2 and u_4) (line 4). Extended embeddings can be safely filtered if violating subgraph isomorphism of G_q (line 5). Binary join can be implemented using GAMMA with a similar process, except that it uses edge extension. Since GAMMA is a framework, we do not build indexing structures for a specific algorithm like SM; instead, we perform pruning and label checking at run-time.

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     $\boldsymbol{Vertex\_Extension}(ET, G_d)$ ;
5     $\boldsymbol{Filtering}(ET, Constraint = G_q)$ ;
6 end
7 output_result($ET$) ;
```

2) Frequent Pattern Mining: FPM uses edge extension. Initially, all length-1 embeddings are recorded in ET and the pattern table PT is empty (line 1 in Algorithm 2). In each iteration, we map all extended embeddings from the last iteration to patterns, append those patterns to PT, and calculate the support of each pattern (line 3). We filter out patterns in PT that do not satisfy the given threshold. The instances of invalid patterns are also removed from ET (line 4). If it is not the last iteration, all embeddings in ET are extended by one edge (line 6). Fig. 2(c) illustrates the above process.

Algorithm 2: Frequent Pattern Mining

```
Input: data graph $G_d$, pattern length limit $l$, minimum support
       $sup_{min}$, map function $mf$.
Output: all frequent patterns.
1 $ET \leftarrow$ all length-1 embeddings, $PT \leftarrow \phi$;
2 foreach $i \in [1,l]$ do
3     $\boldsymbol{PT = PT \cup Aggregation}(ET, mf)$ ;
4     $\boldsymbol{Filtering}(ET, PT, Constraint = sup_{min})$ ;
5     if $i<l$ then
6         $\boldsymbol{Edge\_Extension}(ET, G_d)$ ;
7     end
8 end
9 output_result($PT$) ;
```

IV. Graph Data Storage and Access

We adopt Compressed Sparse Row (CSR) [8], [10], [17] to represent a data graph G_d, which is made up of adjacency lists of all vertices. Both edge-extension and vertex-extension in Section III-B need to access adjacency lists. Therefore, we need to design an efficient access strategy to improve overall performance, especially in out-of-core systems.

GPU device memory is limited. To ensure high write performance, it is necessary to maintain a buffer of some extension results (the last column of the embedding table) in device memory. Unified memory also needs a device buffer. Therefore, there is insufficient space left for graph data in the device when processing large data graphs. As a result, we

[1] The canonical label of a graph is a code that uniquely identifies the graph such that two graphs have the same code if and only if they are isomorphic.

maintain data graph G_d in host memory and propose a self-adaptive host memory access strategy.

As mentioned in Section II-B, unified memory access [25] is friendly to data with good spatial or temporal locality, while zero-copy memory access [26] is suitable for infrequently accessed and isolated data. We use both access methods to exploit both of their advantages. Consequently, we duplicate the CSR of data graph in both unified memory and zero-copy memory. Graph duplication is not a big issue considering the host memory capacity. GAMMA does not build any auxiliary data structures other than structural information and labels to represent graphs. Therefore, the storage of a graph with one billion edges takes only 10 $\sim$ 15 GB.

Adjacency lists are organized in memory pages. The key issue in this section is to determine the access strategy for each requested page p. In GAMMA, embeddings are extended in parallel using the device cores. Before the extension, we know the list of vertices whose adjacency lists will be used. Thus, for each page p, we can calculate how much data in p will be accessed in the next extension. If a large portion of p will be accessed (such as page 2 in Fig. 4), we use the unified memory access to p (multiple threads access the same page p); otherwise, p is accessed by the zero-copy memory (such as page 1 in Fig. 4).

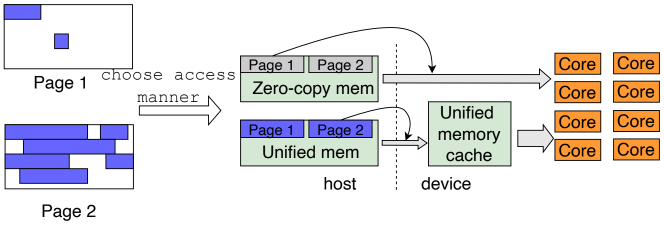


Fig. 4. Two different host memory access methods.

1) Spatial Locality: Existing work [27] shows that serial graph algorithms have poor spatial locality because of their irregular access patterns, resulting in low cache hit rate. However, massive parallel memory accesses enlarge the memory footprint for a period of time, making some pages have good spatial locality. Usually, these pages have high-degree vertices or vertices with some specific labels. If a page p is accessed multiple times by different GPU threads in the same extension, p has good spatial locality and is suitable for unified memory access. The spatial locality due to this parallel access can be used to improve access performance. Intuitively, spatial locality defines how much data in p will be accessed, and this can be formulated as follows.

Definition 4.1 (Spatial Locality): The spatial locality of a page p in the i-th extension is defined by the access quantity of p, i.e.,

$$SpatialLoc_i(p) = \sum_{l(v)\in p \wedge l(v)\in A_i} |l(v)| \times times_i(l(v)) \quad (3)$$

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 in i-th extension.

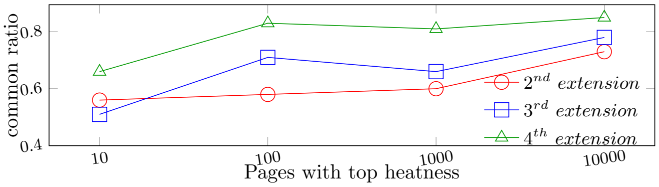


Fig. 5. We take four extensions in SM in different datasets, and show the ratio of duplication of most frequently accessed pages between current extension and past extensions.

2) Temporal Locality: A page p has good *temporal locality* implies that when p is accessed, there is a high probability that it will be accessed again in the near future. Experiments (in Fig. 5) show that extensions have good *temporal locality*. Generally, duplicated hot pages in different extensions take over half of all hot pages, and even reach 70% when we calculate enough pages. Thus, we define the temporal locality as follows:

Definition 4.2 (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.,

$$TempLoc_i(p) = \sum_{j\leq i-1} \sum_{l(v)\in p \wedge l(v)\in A_j} |l(v)| \times times_j(l(v)) \quad (4)$$

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

$TempLoc_i(p)$ is similar to $SpatialLoc_i(p)$, except that it is a summarized parameter telling the historical access frequency of page p. Some pages have good temporal locality in the extensions in GAMMA (shown in Fig. 5). Thus, a page p with large $TempLoc_i(p)$ will be accessed through the unified memory, as p can be cached in device for further extensions.

3) Access Heat: We define the *access heat* for each page p at the i-th extension that combines *spatial locality* and *temporal locality* to model how likely it is for the page to be accessed. We weigh the two factors by the ratio between the total accessed data in the i-th extension and the historical accessed data in the first $(i-1)$ extensions as follows.

Definition 4.3 (Access Heat): The access heat of a page p is defined as follows:

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

where A_j denotes the total accessed data in the j-th extension.

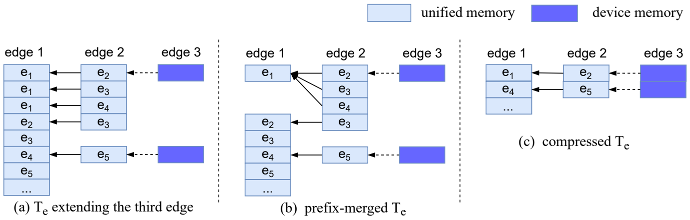
After each extension, $AccHeat_i(p)$ of each page is updated, and they are used to determine memory access method in the following extension: pages accessed through unified memory have buffers in the device so that the maximum number N_u of those pages is determined by the available size of device buffer; N_u hot pages with the largest $AccHeat$ will be accessed by unified memory, while other data will be accessed by zero-copy memory. This self-adaptive method learns hot adjacency lists (or pages) in run-time without introducing too much overhead, and improves overall bandwidth compared with only using zero-copy memory or unified memory.

V. GAMMA: Implementations and Optimization

In this section, we introduce the implementation details and the time complexity of GPM in GAMMA.

A. *Embedding Table*

Data structure. Intermediate results in GPM include many embeddings. Embeddings extended from the same parent share a common prefix. Thus, we can use a prefix-tree to store the embeddings compactly [8], [28]. For example, e-ET T_e in FPM in Fig. 2(c) is extended to the third edge, as shown in Fig. 6(a), and Fig. 6(b) shows an embedding table after merging common prefixes.



(a) T_e extending the third edge

(b) prefix-merged T_e

(c) compressed T_e

Fig. 6. Data structure and data layout of the embedding table.

Some embeddings are invalid after "filtering", and compressing the embedding table will save much space, as in Fig. 6(c). The space compression also provides a better chance for coalesced memory access. However, the compression is ignored in existing GPM frameworks [5], [7]–[9]. Our compression operation has three stages: firstly, the valid and invalid embeddings are marked separately; then, a prefix-scan, which is an efficient operation on GPU, is performed on all marks to obtain new positions of valid embeddings in the compressed embedding table; finally, valid elements are collected in parallel to form the compressed embedding table.

Data layout. The embedding table is stored in column-first fashion: each column of vertex or edge table (e.g., e_1, e_4 in the first column of Fig. 6(c)) is stored consecutively for coalesced reading and writing, and each vertex (or edge) has a pointer to its predecessor in the same embedding. The size of the embedding table may grow exponentially in GPM algorithms. Therefore, it should be resident in host memory. The access to the embedding table is concentrated and continuous, because many embeddings are extended in batches, which have continuous ancestor units since the embedding table is stored in columns. Thus, we use the unified memory for the embedding table. Furthermore, writing results to host memory directly is much slower than writing to GPU device memory. Therefore, we keep a buffer on the device to write extension results, as shown in Fig. 6, and flush them to host memory after the extension of embeddings.

B. *Primitive Optimizations*

The primitives "extension-aggregation-filtering" are convenient for users to implement various GPM algorithms, but their efficient execution over large graphs in an out-of-core GPU system brings new challenges. We discuss them and propose our solutions in the remainder.

Challenge 1: Parallel Write Conflict. When thousands of threads on a GPU are doing parallel extensions, each thread produces an uncertain number of results. As a result, parallel threads do not know the position they should start writing. We refer to this as "parallel write conflict". Most GPU systems, such as Pangolin [8], solve this by doing the same process twice: the first round records the number of results produced by each thread, and the same process is repeated to collect the results. This method solves the write conflict with an additional extension, leading to a severe performance decline. GSI [10] estimates the maximum result set size for each thread and pre-allocates enough space, but the overestimation often causes significant space waste. In a word, existing methods are limited with extra time cost or space cost.

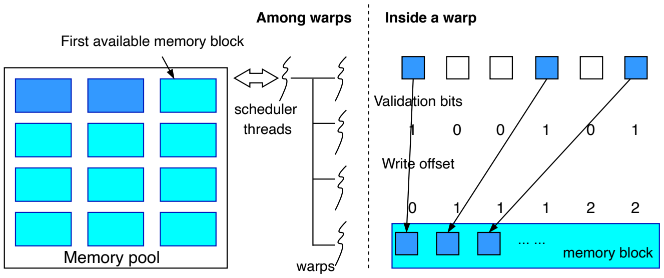


Fig. 7. Dynamic memory allocation.

Optimization 1. To solve the write conflict problem, we design a dynamic memory allocation strategy. The available memory is divided into many memory blocks that form the memory pool. Each warp is assigned a memory block into which it writes the results of embedding extension. When the allocated memory block is full, the warp requests a new one from the memory pool and continues with the extension, as shown in Fig. 7. A scheduler is responsible for the whole memory pool and responds to warp requests. Dynamic allocation solves the write conflict problem among warps. Write conflict among threads within a warp is solved by warp-level prefix scan. Here we choose a warp as the write unit of a memory block, because compared with using thread blocks, the SIMT feature of warp helps solve intra-warp thread conflict at minimum cost; compared with using threads, fewer write units help reduce memory allocation contention and cut down the waste of memory blocks.

The additional time overhead is due to the memory block allocation competition between warps. However, the GPU kernel only has hundreds of active warps, and each warp only asks for a new memory block after it finishes writing the current one. This limits the additional time overhead. The additional space is needed only when the entire process is finished but a warp has not used up its current memory block. In the worst case, hundreds of memory blocks might be wasted. However, in our setting, a memory block is only 8 KB, so this additional storage overhead can be ignored compared with large-scale intermediate results. Thus, our method is both time-efficient and space-saving.

Challenge 2: Duplicate Computation. The second challenge is computational redundancy in the intersection of multiple lists, a common operation in many GPM algorithms such as kCL and SM. The state-of-the-art GPM implemen-

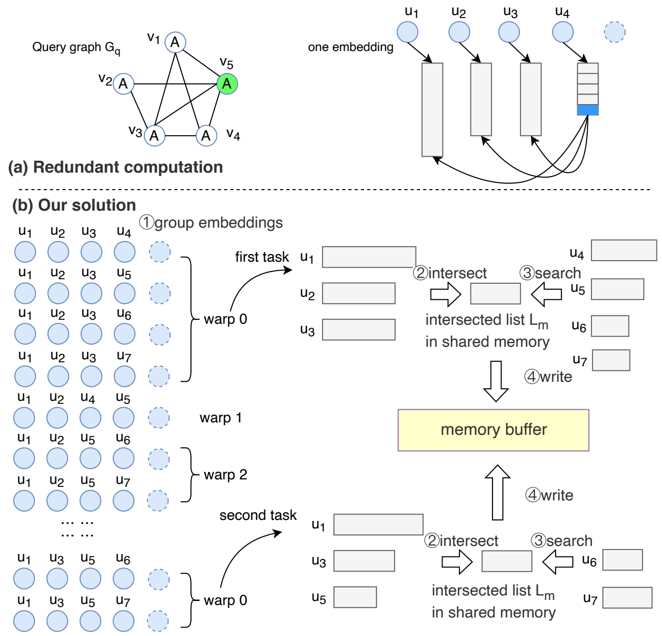


Fig. 8. Redundant computation and our solution.

tations on GPU, such as Pangolin [8], have large amounts of computational redundancy. Consider a query graph G_q and one embedding (u_1, u_2, u_3, u_4) that matches the subquery induced by (v_1, v_2, v_3, v_4) (Fig. 8(a)). The query vertex v_5 to be matched is adjacent to u_1, u_2, u_3 and u_4. Pangolin extends the embedding by enumerating each neighbor of u_4, and searches it in the adjacency lists of u_1, u_2 and u_3. This introduces duplicate computation because those three adjacency lists are accessed and searched multiple times. Furthermore, this computational redundancy is even higher for parallel extension. Consider the multiple embeddings in Fig. 8(b): the first four embeddings are produced by the same parent embedding, therefore they have the same prefix. The naive extension leads to more redundant memory accesses and computation over the adjacency lists of u_1, u_2 and u_3.

To address this problem, we can intersect the adjacency lists of u_1, u_2 and u_3 to get an intersected list L_m, then intersect L_m with the adjacency list of u_4.

Optimization 2. We use *shared memory*, a fast on-chip memory to store pre-intersected lists in order to reduce computational redundancy and accelerate memory access.

Embedding extension is done in four steps in GAMMA (Fig. 8(b)). First, the embeddings are classified into different groups according to their prefixes. For example, the first four embeddings belong to the same group since they share the same prefix (u_1, u_2, u_3), and the next group contains only one embedding in Fig. 8(b). Then, one warp is responsible for extending the embeddings in one group. The intersection of the prefix's adjacency lists produces the intersection list L_m. For example, $L_m = N_v(u_1) \cap N_v(u_2) \cap N_v(u_3)$, where $N_v(u_1)$ denotes the adjacent neighbors of u_1. Finally, in this example, warp 0 intersects the adjacency lists of u_4, u_5, u_6 and u_7 with L_m and writes these results into a warp-level results buffer (memory block), which was discussed in Challenge 1. Once an embedding group extension is completed, warp 0 will move on to the next assigned task, and the results are collected in the same memory block.

This optimization is suitable for BFS-based extension method, where embeddings with the same parent are processed concurrently. The BFS-based method is widely used on GPU, because memory access of neighbor threads is concentrated and coalesced in this manner.

Challenge 3: GPU-based External Sort. Aggregation over the pattern table PT needs to sort the canonical labels of all pattern graphs in PT. However, the size of PT may be beyond the capacity of device memory. Thus, optimizing out-of-core GPU sorting is a challenge. To the best of our knowledge, most GPU-based sorting algorithms, except for two works [29], [30], assume that inputs fit in GPU memory. However, those two methods do not fully utilize GPU parallelism. Thus, we propose an optimized out-of-core GPU sorting algorithm.

Optimization 3. We first partition PT into segments S_i $(i = 1, ..., n)$ such that each segment S_i can be sorted by in-core GPU sorting algorithms [31]. These n sorted segments S_i are written back to the host memory, and merged using the multi-merge algorithm (Algorithm 3).

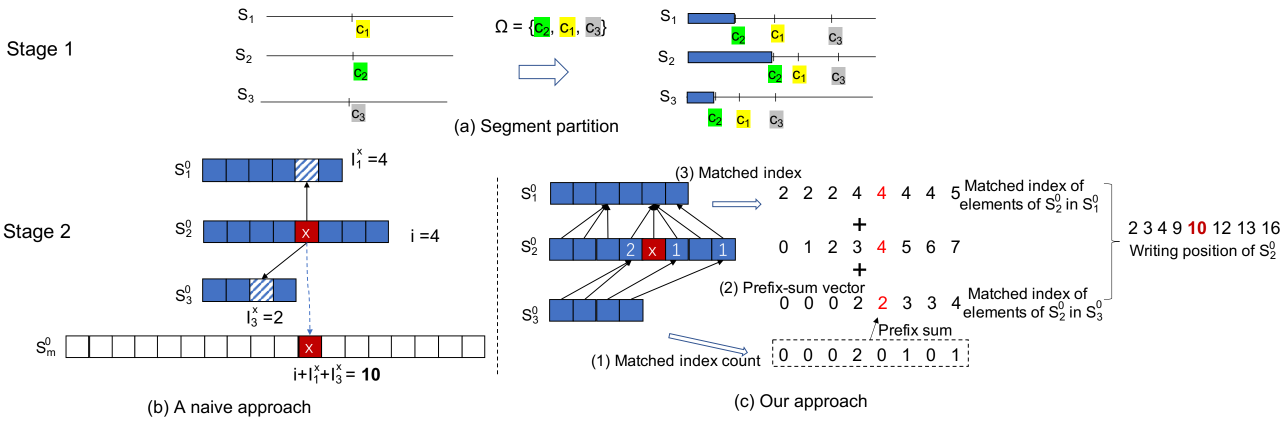
For each segment S_i, its *checkpoints* are defined as the points that divide S_i into partitions of even size, denoted as p_{size}. In the example given in Fig. 9(a), each segment is partitioned into two parts. The set of checkpoints of S_i is denoted as C_i. Algorithm 3 starts by collecting all the checkpoints of S_i $(i = 1, ..., n)$ to get a set Ω (line 2). For each checkpoint $x \in \Omega$, the algorithm finds the *matched index* $\S_i$ in each S_i. Intuitively, the matched index of x over a non-descending sorted segment S_i denotes the largest index $\S_i$ in S_i, where x is no larger than $S_i[\S_i]$. Formally, we have the following definition.

Definition 5.1 (matched index): Given a value x and a sorted segment S_i, the *matched index* of x in S_i, denoted as $\S_i$, is defined as: (1) $0 < \S_i < |S_i|$ if $S_i[\S_i - 1] < x \leq S_i[\S_i]$; or (2) $\S_i = 0$ if $x \leq S_i[0]$; or (3) $\S_i = |S_i|$ if $x > S_i[|S_i| - 1]$.

Finding the matched indices of different checkpoints on different segments can be easily parallelized on GPU.

In this way, each segment S_i is partitioned into $|\Omega| + 1$ lists S_i^o, $o = 0, ..., |\Omega|$ (see Fig. 9(a)). So we divide the task of merging n segments S_i $(i = 1, ..., n)$ into many subtasks of merging short segments (line 4). Our partition method with "checkpoints" and "matched index" assures that each partition size is no larger than p_{size}, otherwise severe workload imbalance may occur. These subtasks can be conducted independently, achieving high parallelism on GPU. Fig. 9(a) gives an example. The first subtask merges all S_i^0 $(i = 1, ..., n)$ (marked in blue), which are smaller than the first checkpoint c_2. Thus, the merged list of all S_i^0 $(i = 1, ..., n)$ should precede the list merging all S_i^1 $(i = 1, ..., n)$.

For explanation, we only discuss how to merge each 0-th list in all S_i $(i = 1, ..., n)$ (lines 7-23). Fig. 9(a) highlights these on the three sorted lists, denoted as S_1^0, S_2^0, and S_3^0, which are merged into a sorted list S_m^0. A naive solution works as follows. Consider each element x in S_2^0 (assume that $S_2^0[i]$=x):



(a) Segment partition

(b) A naive approach

(c) Our approach

Fig. 9. Two stages of multi-merge.

we search the matched index of x in all other lists (i.e., S_1^0 and S_3^0 in Fig. 9(b)), denoted as I_1^x and I_3^x, respectively. We can infer that the final index of x in S_m^0 is i + I_1^x + I_3^x, as illustrated in Fig. 9(b). We perform the same process for each element in all sorted lists S_i^0 $(i = 1, ..., n)$ to locate elements in the merged list S_m^0, which can be parallelized.

Algorithm 3: Multi-merge Kernel

```
   Input: sorted list set $S$ ($|S|$ = n).
   Output: One Merged List.
1  /* block-wise spliting lists*/;
2  $check\_points$ ← get_check_points($S$);
3  /* block-wise dividing tasks*/;
4  $subtask\_set$ ← divide($S$, $check\_points$);
5  /* warp-wise merging short lists*/;
6  foreach $i^{th}$ $subtask \in subtask\_sets$ do
7      $S_1^i$,..., $S_n^i$, $global\_off$ ← get_subtask($subtask\_set$, $i$);
8      $writing\_pos$[][] ← initial_n_writing_pos();
9      /* handling each ($S_j^i$,$S_k^i$) pair*/;
10     foreach $j \in [1,n]$ do
11         foreach $k \in [1,j)$ do
12             $matched\_idx$[], $matched\_cnt$[] ← zeros();
13             /* thread-wise searching for matches*/;
14             foreach $p \in [0,|S_j^i|)$ do
15                 $pos$ ← search_for_match($S_j^i[p]$, $S_k^i$);
16                 $matched\_idx[p]$ ← $pos$;
17                 $matched\_cnt[pos]$ += 1;
18             end
19             $writing\_pos[j]$.vector_add($matched\_idx$);
20             prefix_sum($matched\_cnt$);
21             $writing\_pos[k]$.vector_add($matched\_cnt$);
22         end
23     end
24     /* thread-wise writing merged results $S_m^i$*/;
25     foreach $S_j^i \in S_1^i$,...,$S_n^i$ do
26         parallel_writing($S_j^i$, $writing\_pos[j]$, $global\_off$);
27     end
28 end
```

Some redundant search, e.g., searching elements of S_2^0 over S_3^0, can be avoided. We can define an order of these short segments (i.e., S_1^0, S_2^0 and S_3^0 in Fig. 9(c)) and only search elements of S_j^0 over S_k^0, where $j > k$ (lines 14-18). Assume that all elements in S_3^0 find their matched indices in S_2^0, we count all matched indices at each position of S_2^0 to obtain the vector [0,0,0,2,0,1,0,1]. The prefix-sum over this vector generates vector [0,0,0,2,2,3,3,4] (lines 20-21), which denotes the matched indices of elements in S_2^0 over S_3^0. For the i^{th} element x in S_2^0, the i^{th} element in the prefix-sum vector denotes the matched index I_3^x over S_3^0. Thus, we avoid searching x over S_3^0. Fig. 9(c) demonstrates how to compute writing positions of all elements in S_2^0, where the 4-th element x is highlighted in red. Prefix-sum is an efficient operation on GPU, thus we can save about half of the workloads.

C. Complexity Analysis

The complexity of GPM tasks is primarily due to combinatorial enumeration and isomorphism check, and the most time-consuming stage is the final extension because of its exponentially increased intermediate results size [8]. Here we give the worst-case complexity analysis.

Considering SM algorithm on an input graph G with n vertices and the maximum embedding size of k, the maximum degree in G is denoted as d_{max}. There are up to $O(n^{k-1})$ size-$(k-1)$ embeddings (partial matches) in the embedding table before the final extension. For each size-$(k-1)$ partial match p, assume that we extend the embedding from one data vertex in p. Obviously, there are up to $O(n^{k-1}d_{max})$ possible new candidate embeddings. For each new candidate match p', we need to check adjacency of the new extended vertex v $(v = p' - p)$ with the other $k-2$ vertices in size-$(k-1)$ partial match p. The adjacency check is done using binary search over the adjacency lists. Thus, the overall complexity is $O(n^{k-1}d_{max}(k-2)log(d_{max}))$. That is the complexity of naive combinatorial enumeration as implemented in Pangolin [8].

Our Optimization 2 groups embeddings to avoid redundancy. In the last extension, all size-$(k-1)$ embeddings can be grouped by sharing size-$(k-2)$ embeddings as parents. Generally, there are up to $O(n^{k-2})$ embedding groups. In processing each group, we first intersect the adjacency lists of $k-2$ prefix vertices, whose complexity is $O((k-2)d_{max})$ for each group. As analyzed in the last paragraph, there are

$O(n^{k-1}d_{max})$ new size-k candidate embeddings p'. For each candidate embedding, we only need to check the adjacency of the new extended vertex with regard to the pre-intersected list, whose time complexity is $O(log(d_{max}))$, since the pre-intersected list length is $O(d_{max})$. Therefore, the complexity of combinatorial enumeration in GAMMA is $O(n^{k-2}(k-2)d_{max} + n^{k-1}d_{max}log(d_{max}))$, which is less than that of Pangolin because of the grouping operation.

The complexity of isomorphism test for each new embedding is $O(e^{\sqrt{klogk}})$ [32]. Considering both combinatorial enumeration and isomorphism check, the worst-case complexity is $O(n^{k-2}(k-2)d_{max} + n^{k-1}d_{max}(log(d_{max}) + e^{\sqrt{klogk}}))$. Other GPM algorithms can be analyzed similarly.

Assuming there are w warps in the device, the parallel complexity is $O(\frac{n^{k-2}(k-2)d_{max}+n^{k-1}d_{max}(log(d_{max})+e^{\sqrt{klogk}})}{w})$ in which the tasks of each warp are independent. Thread parallelism inside warps is affected by memory access and thread divergence, which can further reduce the parallel complexity.

VI. Experiments

A. Experimental Setting

Infrastructure. We use the CUDA-9.0 toolkit and GCC 5.3.0 to compile all codes with -O3 option. All experiments are carried out on a Linux server with Intel Xeon E5-2640 CPU, a 32-core processor and 380 GB of host memory. It also has an NVIDIA Tesla V100 with 16 GB global memory.

TABLE II
Datasets Infos

dataset	nodes	edges	types
cit-Patent(CP)	6M	17M	citation
com-lj(CL)	4M	34M	social
com-orkut(CO)	3M	117M	social
email-Euall(EA)	265K	729K	email
email-Enron(ER)	37K	368K	email
com-lj×8(CL×8)	32M	467M	synthetic
soc-Live×5(SL×5)	24M	481M	synthetic
uk2005(UK)	39M	1.6B	web
it2004(IT)	41M	2.1B	web
twitter_rv(TW)	62M	2.4B	social

Datasets. We use a number of real graphs with varying sizes from different domains. To test the the scalability of GAMMA to large graphs, we scale up *soc-Live* and *com-lj* by 5× and 8× using graph upscaling technique [33]; we also use billion-scale real-world graphs. Table II lists all datasets.

Comparative evaluation. We use subgraph matching (SM), frequent pattern mining (FPM) and k-clique (kCL) workloads to compare GAMMA[2] with the state-of-the-art methods:

Pangolin [8] is the only GPM framework on GPU. It provides an API for users to apply application-specific pruning to reduce enumeration space, resulting in comparable performance to specific implementations. We adopt both its GPU implementation and single-thread implementation as baselines to evaluate GAMMA's performance.

Peregrine [16] is a state-of-the-art GPM framework on CPU, which uses multi-threads to improve performance and is superior to other GPM systems, including Arabesque [5], Rstream [6] and Gminer [34]. Therefore, we use Peregrine as the multi-thread CPU baseline.

We also use some task-specific implementations to demonstrate that GAMMA makes it easier to implement graph mining algorithms without sacrificing performance. We compare with GSI [10], a state-of-the-art subgraph matching algorithm on GPU, for subgraph matching. It uses "prealloc-combine" method to avoid joining-twice. It also introduces a GPU-friendly data structure to improve the joining phase in SM. Since existing GPU algorithms do not have good support for FPM on large graphs, we use FPM implementation in GraphMiner [35], which is a multi-core CPU-based graph algorithm library[3] that combines several state-of-the-art GPM designs [8], [36], [37].

B. Memory Usage

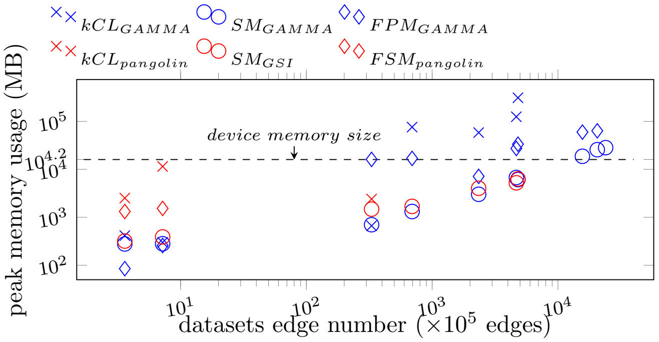


Fig. 10. Peak memory usage.

The peak memory usage of GAMMA and other GPU-based GPM implementations, including host memory and device memory, is shown in Fig. 10. GAMMA uses less memory than other GPM implementations for a given input graph, because of our compression of the embedding table. There are many cases in GAMMA where memory usage exceeds available device memory. The maximum used space reaches 310 GB in some large graphs. In-core GPM algorithms only use device memory and they cannot run on large graphs.

Fig. 10 also shows that SM uses less memory than FPM, and FPM uses less memory than kCL; because SM has the most pruning conditions, and kCL has few pruning conditions. As a result, the maximum graph size in SM of GAMMA is the largest among the three algorithms in our experiments, and that of kCL is the smallest.

C. Comparative Evaluation

GAMMA's comparative evaluation with the baselines for each of the workloads is discussed below.

K-clique. The experimental results of GAMMA compared with state-of-the-art works for kCL are shown in Fig. 12. "Pangolin-ST" denotes single-thread verison of Pangolin, and "Pangolin-GPU" denotes GPU verison of Pangolin. Some

[2] Our codes are released on github: https://github.com/pkumod/GAMMA.

[3] We call it a library rather than a framework because it implements these graph algorithms separately, without abstracting primitives or uniform processing framework like GAMMA.

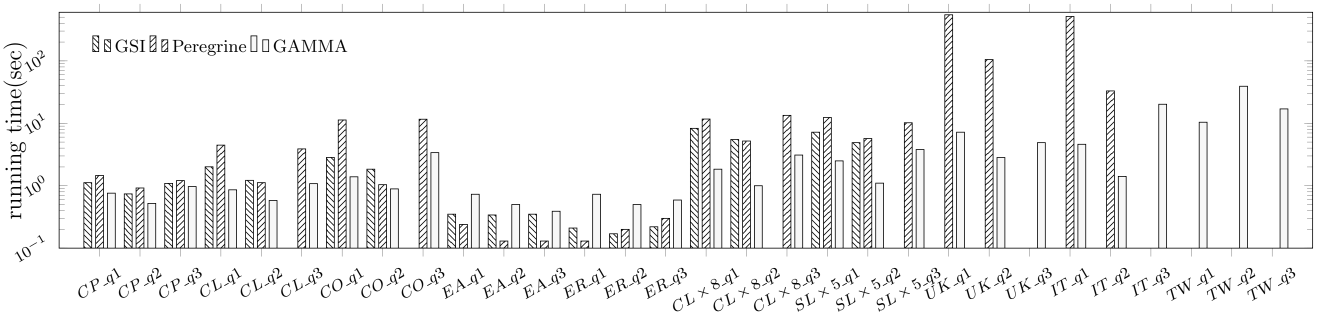


Fig. 11. Performance of SM.

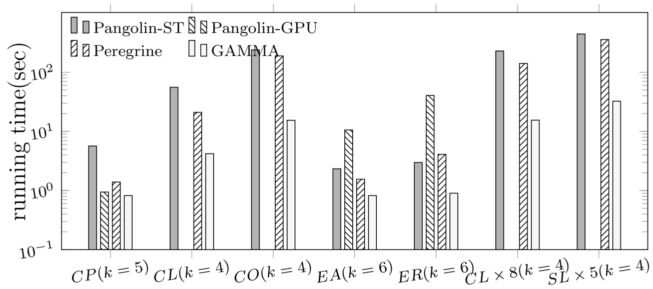


Fig. 12. Performance of kCL.

works crash on some of the datasets, and we omit those cases in the figure. GAMMA has good scalability for all datasets. It has better performance than "Pangolin-GPU" and Peregrine: it achieves an average of 67.6% and 73.9% speedup, respectively.

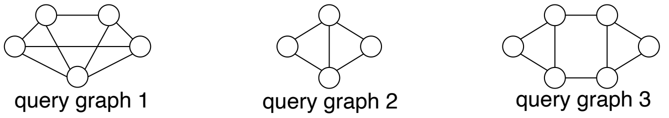


Fig. 13. Query graphs in SM.

SM. Fig. 11 reports the running times of GAMMA, GSI and Peregrine for three SM queries (as shown in Fig. 13) on each dataset. We do not compare with Pangolin or GraphMiner, since they do not have subgraph matching implementations. GAMMA performs much better than GSI and Peregrine on all large graph datasets except for two small datasets (EA and ER), achieving 50.6% speedup over GSI and 70.5% speedup over Peregrine on all datasets. For small datasets, the preparation of host memory usage in GAMMA accounts for a large portion of the total running time. Thus, it is slower than the in-core GPU implementation of GSI and the CPU-based Peregrine. GSI and Peregrine crash on some datasets in our experiments, which we omit in Fig. 11.

FPM. We compare GAMMA with GraphMiner, Peregrine and Pangolin in FPM. As shown in Fig. 14, GAMMA has great scalability advantages compared with other works: it can process billion-scale graphs, while other methods meet crashes in large datasets. GAMMA has 86.1% and 73.8% performance improvement compared with "Pangolin-ST" and "Pangolin-GPU", respectively. It achieves an average of 50.6% speedup compared with Peregrine. Although GraphMiner implements a specific FPM algorithm, GAMMA still has slightly better performance, achieving 24.7% performance improvements.

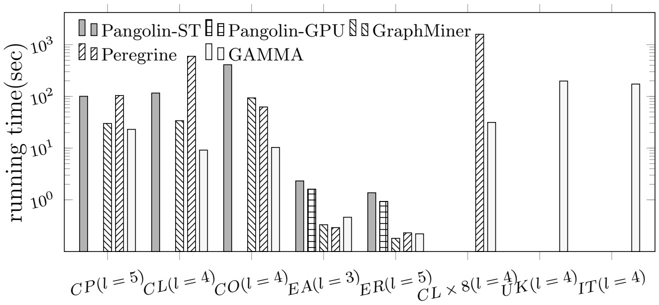
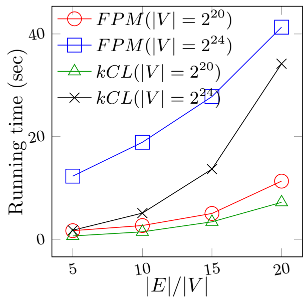


Fig. 14. Performance of FPM compared with state-of-the-art works.

GAMMA's performance superiority is due to the optimization of *aggregation* primitive (Optimization 3 in Section V). This alleviates the device memory limit and allows the aggregation for huge embedding tables. Compared with the four baselines, the optimized three-phase processing framework of GAMMA guarantees performance superiority in FPM.



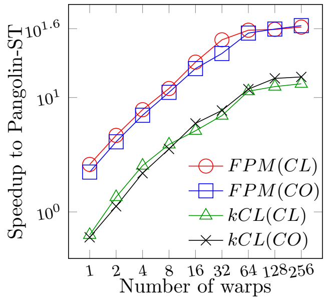


Fig. 15. Different graph densities. Fig. 16. Different number of warps.

D. Scalability

We have evaluated the scalability of GAMMA ranging from small graphs to billion-scale graphs in Section VI-C. As shown in Fig. 11 and Fig. 14, GAMMA can process much larger graphs than other works, and scales well with graph size.

Next, we focus on the scalability of graph density and warp number. We generate kronecker graphs [38] with different numbers of vertices and different graph densities. As shown in Fig. 15, GAMMA has good scalability with respect to graph density, and its running times increase approximately linearly as the graph density increases.

Warp is the basic unit for memory access and thread collaboration on GPU. We present the performance of GAMMA under different numbers of warps in Fig. 16, where we

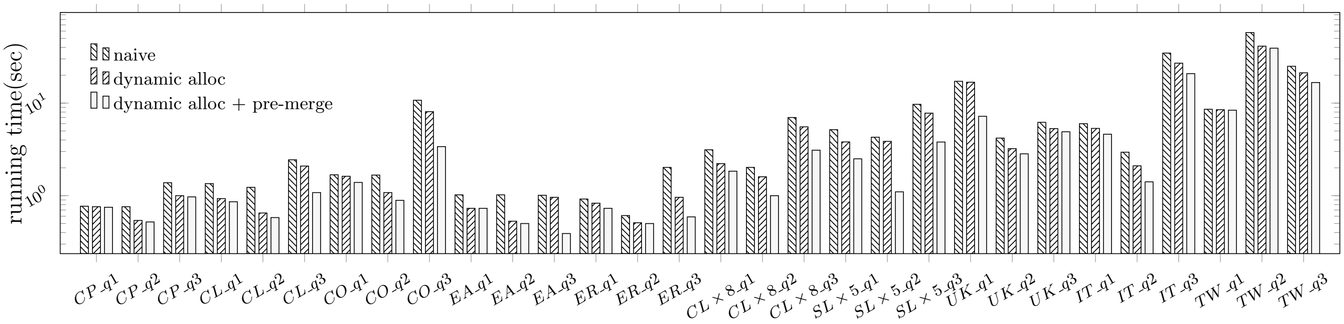


Fig. 17. The effect of optimizations on SM.

use the performance of "Pangolin-ST" as a baseline, and plot GAMMA's normalized speedup. GAMMA outperforms "Pangolin-ST" with one warp or two warps, and has approximately linear performance improvements as the warp number increases.

E. Evaluation of Primitive Optimizations

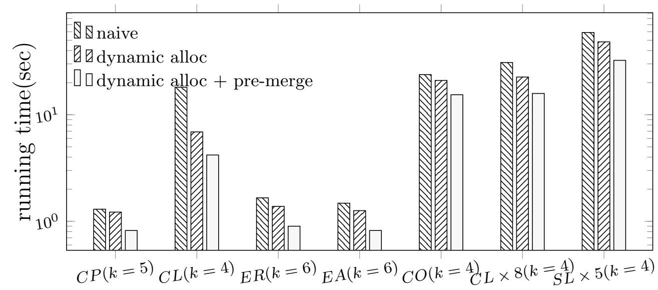


Fig. 18. The effect of our optimizations on kCL.

In this subsection, we evaluate the effectiveness of the three optimizations discussed in Section V-B. The first two optimizations are related to the "extension" primitive. We design a dynamic memory allocation strategy, which is denoted as "dynamic-alloc" in the following figures. We also avoid duplicate computation by grouping embeddings with the same prefix. This optimization is denoted as "pre-merge" in the figures. As a baseline, "naive" method does not have either of those two optimizations. Note that the third optimization for out-of-core multiple lists intersection (denoted as "multimerge-opt") is only involved in FPM to compute the support of patterns. Therefore, we evaluate the first two optimizations in SM and kCL in Fig. 17 and 18, respectively.

Fig. 17 and 18 show that both "dynamic-alloc" and "pre-merge" are effective in improving the performance significantly, especially in some large graphs. "dynamic-alloc" helps speed up the naive approach by 21.7% on average, and "pre-merge" further achieves 25.4% performance improvements.

TABLE III
FPM PERFORMANCE OF DIFFERENT SORTING METHODS.

running time(sec)	cpusort	xrt2sort	multimerge	multimerge+opt
CL×8	42.37	33.12	35.87	31.14
SL×5	40.05	32.23	33.8	30.85

Sorting a list that exceeds device memory is an essential operation for our aggregation primitive, and Optimization 3 reduces the computation time of this operation (which is called "multimerge+opt"). Existing works of out-of-core GPU sort usually involve considerable CPU processing [29], [30]. Thus, they cannot achieve the maximum parallelism. We compare "multimerge+opt" with "cpusort", a popular sorting method implemented by Thrust [39] on CPU, and "xtr2sort" [30], a state-of-the-art out-of-core sort implementation on GPU that replaces merging by data rearrangement and sorting twice. We also include "multimerge", which searches each item of segments over other segments (see Fig. 9(b)). Not all datasets need the external sort of the pattern table PT, and we give the performance of two datasets as examples in Table III, in which our approach runs the fastest compared with different baselines.

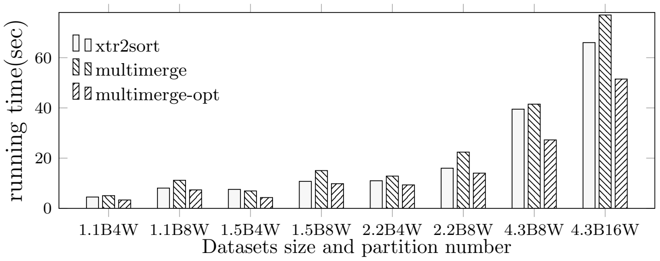


Fig. 19. Effect of Optimization 3 on multi-merge.

To further analyze the effectiveness of Optimization 3 on the sorting process alone, we generate 64-bit value sets of different sizes and perform multi-merge with different methods. CPU-based sorting is much worse than other GPU-based methods, as shown in Table III, and we do not plot its results. In Fig. 19, labels of the horizontal axis denote tasks. For example, "4.3B8W" indicates that 8-way multi-merge is performed on 4.3 billion 64-bits values. From Fig. 19, we conclude that this optimization achieves 34.2% speedup over the naive implementation, and 20.9% speedup over xtr2sort.

F. Evaluation of Hybrid Memory Access

We evaluate GAMMA's memory access determination strategy over all three GPM workloads. The results are shown in Fig. 20. We use unified memory alone and zero-copy memory alone as baselines. As discussed earlier, host memory accesses vary a lot, so neither single access method alone works well. GAMMA's combined memory access method achieves 47.4% speedup over only using unified memory and 51.0% speedup over only using zero-copy memory.

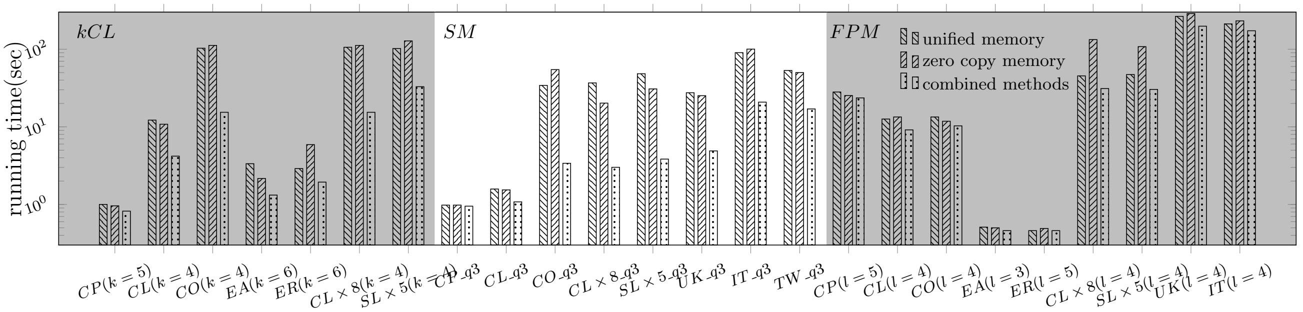


Fig. 20. The effect of our memory access mode.

Note that the performance gains brought about by primitive optimizations and the hybrid access strategy are orthogonal, because the former are algorithmic-level designs while the latter is an optimization on the underlying memory access.

VII. Related Work

A. Existing GPM Frameworks

Existing GPM frameworks are designed on disk-involved platforms [6], [7], [40], distributed systems [5], [9], [34], multi-core CPU systems [16] and GPU [8]. These works distinguish graph mining algorithms (such as kCL, SM and FPM) from graph traversal algorithms, and build universal solutions for them.

Kaleido [7] is a single-machine GPM system. It uses a lightweight checking strategy to solve labeled graph isomorphism problems. Arabesque [5] is a distributed system that defines a high-level filter-process computational model. Rstream [6] is a single-machine GPM system based on X-stream [41]. Peregrine [16] is a pattern-aware multi-core GPM system on CPU, and it manages to reduce unnecessary computations by carefully designing exploration plans.

Pangolin [8] is the only GPM framework on GPU. It gives some optimizations in subgraph isomorphism check, reducing memory usage and exploiting data locality. However, since Pangolin is an in-core system that only uses device memory, it cannot process GPM tasks on even moderate-size graphs.

B. Specific Graph Algorithms on GPU

There are many existing specific graph pattern mining algorithms on GPU, including triangle counting [17], [19], [42], [43] and subgraph matching [10], [14], [15]. Related works of FPM and kCL [44] on GPU are much fewer than those of the first two algorithms, because a large number of intermediate results is not suitable for GPU.

There are some specific algorithms of large graphs on GPU [15], [17], [20]. They all adopt dedicated methods for graph partition or reorganization, which do not work for all algorithms and bring about extra cost.

C. Host Memory Access on GPU

Basically, there are two methods for GPU to process graphs larger than device memory. The first one is explicit data transfer [17], [19], [20]: the required data are reorganized and transferred to device memory in batches, then GPU can directly access data on device. In this condition, data reorganization is time-consuming; this also leads to extra data transfer and low GPU utilization. The second one is on-demand memory access [18], [25] using unified memory and zero-copy memory. This method takes device memory and host memory as unified memory space, and has significant advantages in the simplicity of programming, thus more suitable to design frameworks. Many works spend efforts on improving the performance of unified memory or zero-copy memory by compressing graphs [45], reordering graphs [25], [45], coalesced and aligned memory access [26], or hardware-level schedules [46], [47]. To the best of our knowledge, GAMMA is the first work to propose a hybrid access strategy based on analytic model in a framework, which is a significant contribution of our work.

VIII. Conclusions

In conclusion, we present GAMMA, a GPM framework on GPU for large graphs, which hides implementation details from users and provides flexible and effective primitives. To the best of our knowledge, it is the first framework to support GPM on large graphs on GPU. We design data structures resident in both host memory and device memory, and provide self-adaptive host memory access methods. Therefore, GAMMA can cope with a large number of intermediate results. Processing large graphs on GPU brings about some challenges, and we give three optimizations to improve performance. Extensive experiments show that GAMMA outperforms state-of-the-art GPM frameworks and some dedicated graph algorithms on GPU.

Acknowledgment

This work was supported by NSFC under grant 61932001 and U20A20174. Özsu's research was supported by Natural Sciences and Engineering Research Council (NSERC) of Canada. Lei Zou is the corresponding author of this work.

References

[1] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis, "Frequent substructure-based approaches for classifying chemical compounds," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, no. 8, pp. 1036–1050, 2005.

[2] W.-T. Chu and M.-H. Tsai, "Visual pattern discovery for architecture image classification and product image search," in *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*, 2012, pp. 1–8.

[3] L. Wu and H. Liu, "Tracing fake-news footprints: Characterizing social media messages by how they propagate," in *Proceedings of the eleventh ACM international conference on Web Search and Data Mining (WSDM)*, 2018, pp. 637–645.

[4] W. Eberle, J. Graves, and L. Holder, "Insider threat detection using a graph-based approach," *Journal of Applied Security Research*, vol. 6, no. 1, pp. 32–81, 2010.

[5] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 425–440.

[6] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine," in *Proceeding of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI))*, 2018, pp. 763–782.

[7] C. Zhao, Z. Zhang, P. Xu, T. Zheng, and J. Guo, "Kaleido: An efficient out-of-core graph mining system on a single machine," in *Proceeding of the 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 673–684.

[8] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on cpu and gpu," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 8, pp. 1190–1205, 2020.

[9] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2019, pp. 1357–1374.

[10] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "GSI: GPU-friendly subgraph isomorphism," in *Proceedings of IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1249–1260.

[11] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 716–727.

[12] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 7, pp. 517–528, 2014.

[13] Y.-W. Wei, W.-M. Chen, and H.-H. Tsai, "Accelerating the bron-kerbosch algorithm for maximal clique enumeration using gpus," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 9, pp. 2352–2366, 2021.

[14] L. Wang, Y. Wang, and J. D. Owens, "Fast parallel subgraph matching on the gpu," in *Proceeding of the International Symposium on High Performance Distributed Computing (HPDC)*, 2016.

[15] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020, pp. 1067–1082.

[16] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: a pattern-aware graph mining system," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[17] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018, pp. 171–182.

[18] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016, pp. 447–461.

[19] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li *et al.*, "Trust: Triangle counting reloaded on gpus," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 11, pp. 2646–2660, 2021.

[20] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–16.

[21] S. Sun and Q. Luo, "Subgraph matching with effective matching order and indexing," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 34, no. 1, pp. 491–505, 2020.

[22] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, no. 3, pp. 217–228, 2016.

[23] A. Mhedhbi, C. Kankanamge, and S. Salihoglu, "Optimizing one-time and continuous subgraph queries using worst-case optimal joins," *ACM Transactions on Database Systems (TODS)*, vol. 46, no. 2, pp. 1–45, 2021.

[24] Z. Zeng, J. Wang, and L. Zhou, "Efficient mining of minimal distinguishing subgraph patterns from graph databases," in *Proceeding of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2008, pp. 1062–1068.

[25] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 7, pp. 1119–1133, 2020.

[26] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 2, pp. 114–127, 2020.

[27] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016, pp. 1813–1828.

[28] J.-I. Aoe, K. Morimoto, and T. Sato, "An efficient implementation of trie structures," *Software: Practice and Experience*, vol. 22, no. 9, pp. 695–721, 1992.

[29] M. Gowanlock and B. Karsin, "Sorting large datasets with heterogeneous cpu/gpu architectures," in *Proceeding of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 560–569.

[30] H. Sato, R. Mizote, S. Matsuoka, and H. Ogawa, "I/o chunking and latency hiding approach for out-of-core sorting acceleration using gpu and flash nvm," in *Proceeding of the International Conference on Big Data (Big Data)*, 2016, pp. 398–403.

[31] D. P. Singh, I. Joshi, and J. Choudhary, "Survey of gpu based sorting algorithms," *International Journal of Parallel Programming*, vol. 46, no. 6, pp. 1017–1034, 2018.

[32] L. Babai, W. M. Kantor, and E. M. Luks, "Computational complexity and the classification of finite simple groups," in *Proceeding of the Annual Symposium on Foundations of Computer Science (SFCS)*, 1983, pp. 162–171.

[33] H. Park and M.-S. Kim, "Evograph: An effective and efficient graph upscaling method for preserving graph properties," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018, pp. 2051–2059.

[34] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: an efficient task-oriented graph mining system," in *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys)*, 2018, pp. 1–12.

[35] X. Chen, "Graphminer," https://github.com/chenxuhao/GraphMiner.

[36] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: a pattern-aware accelerator for graph pattern mining," in *Proceeding of the Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 581–594.

[37] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2021, pp. 378–391.

[38] https://github.com/graph500/graph500.

[39] https://thrust.github.io.

[40] D. Mawhirter and B. Wu, "Automine: harmonizing high-level abstraction and high performance for graph mining," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 509–523.

[41] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 472–488.

[42] L. Hu, N. Guan, and L. Zou, "Triangle counting on gpu using fine-grained task distribution," in *Proceeding of the International Conference on Data Engineering Workshops (ICDEW)*, 2019, pp. 225–232.

[43] A. Yaşar, S. Rajamanickam, J. W. Berry, and Ü. V. Çatalyürek, "A block-based triangle counting algorithm on heterogeneous environments," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 2, pp. 444–458, 2021.

[44] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, “Parallel k-clique counting on gpus,” in *Proceedings of the 36th ACM International Conference on Supercomputing (ICS)*, 2022, pp. 1–14.

[45] P. Gera, “Overcoming memory capacity constraints for large graph applications on gpus,” Ph.D. dissertation, Georgia Institute of Technology, 2021.

[46] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, “A framework for memory oversubscription management in graphics processing units,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 49–63.

[47] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, “Towards high performance paged memory for gpus,” in *Proceeding of the International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 345–357.