The VLDB Journal (2025) 34:6
https://doi.org/10.1007/s00778-024-00883-8

REGULAR PAPER

# A graph pattern mining framework for large graphs on GPU

Lin Hu[1] · Yinnian Lin[1] · Lei Zou[1] · M. Tamer Özsu[2]

Received: 10 February 2024 / Revised: 15 September 2024 / Accepted: 21 September 2024


**Abstract**
Graph pattern mining (GPM) is an important problem in graph processing. There are many parallel frameworks for GPM, many of which suffer from low performance. GPU is a powerful option for accelerating graph processing, but 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, that makes full use of host memory to process large graphs. GAMMA adopts a self-adaptive implicit host memory access approach to achieve high bandwidth, which is transparent to users. It 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, as well as optimizations to perform set intersections since they are widely used in GPM. Experimental results show that GAMMA scales better with graph size over the state-of-the-art approaches—by an order of magnitude—and is also faster than existing GPM systems.

**Keywords** Graph pattern mining · Large graphs · GPU

## 1 Introduction

The importance of graph algorithms in many fields is well-recognized: chemical engineering [15], social networks [14, 57] and financial markets [17]. Significant attention has been paid to graph pattern mining (GPM) tasks that discover graph patterns satisfying some criteria [12, 16, 50, 51, 63]. This class of workloads involves subgraph matching (SM) [59], frequent pattern mining (FPM) [3, 18] and k-clique (kCL) computation [55]. 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 [63]. 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 [22, 52, 59] and FPM [3, 18]. 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 [12, 16, 27, 50], 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 [50], a state-of-the-art GPM framework, spends 1.65 h 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

Lin Hu and Yinnian Lin have contributed equally to this work.

✉ Lei Zou
zoulei@pku.edu.cn

Lin Hu
hulin@pku.edu.cn

Yinnian Lin
linyinnian@pku.edu.cn

M. Tamer Özsu
tamer.ozsu@waterloo.ca

[1] Peking University, Beijing, China

[2] University of Waterloo, Waterloo, Canada

Published online: 05 December 2024

massive parallelism with a large memory bandwidth compared to a CPU, making it suitable for GPM. Most existing GPU-based works focus on designing specific GPM algorithms [26, 55, 59] rather than a comprehensive framework. To the best of our knowledge, Pangolin [12] 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 [26, 29, 41, 45] partition graphs and explicitly transfer them to GPU for processing. However, this approach requires task-specific partitioning strategies, and incurs redundant memory transfer and extra data reorganization costs. 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 host memory 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. Our proposed data layout solution can also be generalized to multi-GPU architectures: the host accommodates main data structures, and GPUs share nothing. Therefore, GPUs work independently, and the number of GPUs can scale up conveniently.

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 Sect. 4). 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.

Set intersection is widely used in many GPM algorithms [23]. We propose scenario-specific optimizations to further reduce GPM's computational complexity. Set intersection in GPM has different features: the number of lists to be intersected are different; the length of those lists vary a lot; sometimes a single list needs to be intersected with a number of lists multiple times. We take those features into consideration, and give different strategies to improve the performance.

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 1). 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 novel 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. This memory usage strategy is also convenient to scale up to multi-GPU architecture.
- We propose three optimizations to existing GPM framework primitives based on the GPU architecture and graph mining tasks. These optimizations target large graphs.
- According to the features of set intersections in GPM, we propose different optimizations to improve their performance in this scenario.
- We conduct extensive experiments. The results show that GAMMA has great improvements in scalability and performance compared with state-of-the-art works.

**Table 1** Categories of different graph mining works

| | GPU in device | Out of device | CPU |
|---|---|---|---|
| Frameworks | Pangolin [12] | GAMMA | Peregrine [27], Kaleido [63], Arabesque [50], Fractal [16] |
| Specific algos | GSI [59] | Guo et al. [22] | Sun et al. [49] |

## 2 Related work

GPU-based graph computing has attracted considerable attention in both academia and industry [12, 34]. Generally, there are two different graph computing workloads. One is the traversal-based, such as BFS [37] and shortest path computation [33], while the other one is called graph pattern mining (GPM) tasks including subgraph matching (SM), frequent subgraph pattern mining (FPM) and k-clique (kCL) computation. One fundamental difference between them is that the latter always generates a large number of intermediate results while the former's space complexity is linear with respect to the original graph size. In this work, we focus on GPU-based GPM computations. As mentioned earlier, there are two approaches. One is to design and optimize specific GPM algorithms and the other one is to develop a generic GPM frameworks. We review them briefly in the following subsections. We also discuss host memory access and multi-GPU solution for graph computing in Sect. 2.3 and 2.4, respectively, since GAMMA is an out-of-core system and scales to multi-GPU architecture.

### 2.1 Specific GPM algorithms on GPU

There are many existing specific GPM algorithms on GPU, including triangle counting [24, 26, 41, 58] and subgraph matching [22, 52, 59]. Related works of FPM and kCL [4] 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 [22, 26, 45]. They all adopt dedicated methods for graph partition or reorganization, which do not work for all algorithms and bring about extra cost.

### 2.2 Existing GPM frameworks

Existing GPM frameworks are designed on disk-involved platforms [35, 51, 63], distributed systems [8, 16, 50], multi-core CPU systems [27] and GPU [12]. These works distinguish graph mining algorithms (such as kCL, SM and FPM) from graph traversal algorithms, and build universal solutions for them.

Kaleido [63] is a single-machine GPM system. It uses a lightweight checking strategy to solve labeled graph isomorphism problems. Arabesque [50] is a distributed system that defines a high-level filter-process computational model. Rstream [51] is a single-machine GPM system based on X-stream [44]. Peregrine [27] is a pattern-aware multi-core GPM system on CPU, and it manages to reduce unnecessary computations by carefully designing exploration plans. Peregrine [27] 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 [50], Rstream [51] and Gminer [8]. Therefore, we use Peregrine as the multi-thread CPU baseline.

Pangolin [12] is the first GPM framework on GPU and incorporates some optimizations in subgraph isomorphism check to reduce memory usage and exploit data locality. However, since Pangolin is an in-core system that only uses GPU memory, it cannot process GPM tasks on even moderate-size graphs. PBE [22] is a subgraph enumeration solution on GPUs, which divides large graphs into partitions that can fit into GPU memory to scale to enormous graphs beyond GPU memory. PBE processes one partition at-a-time and proposes a method to enumerate matched subgraphs across multiple partitions. We include PBE in our evaluation on kCL and SM. $G^2$Miner [10] is the first to run a GPU-based GPM framework on multiple GPUs, which is an optimization of Pangolin by combining breadth-first search and depth-first search to achieve a balance between intermediate result size and parallelism. $G^2$Miner also proposes a series of GPU-related optimization techniques and a scheduling policy for multiple GPUs.

The most up-to-date GPM framework on GPUs is GraphSet [47]. It focuses on eliminating control flows and reducing computation overhead in user-defined GPM algorithms. GraphSet accelerates these algorithms by transforming control flows and redundant computation into set-based operations. GraphSet has the best computational performance in existing systems but can only support in-core processing and fails to support graphs larger than GPU memory.

We experimentally compare our framework (GAMMA) with these existing solutions in Sect. 7. Generally, GAMMA can support larger graphs and also achieve better perfor-

mance in GPM tasks. More experimental results are given in Figs. 18, 20, 23, 37, and 38.

### 2.3 Host memory access on GPU

There are two methods for GPU processing of graphs larger than device memory. The first one is explicit data transfer [26, 41, 45]: 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, and also leads to extra data transfer and low GPU utilization. The second one is on-demand memory access [20, 29] 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 have focused on improving the performance of unified memory or zero-copy memory by compressing graphs [19], reordering graphs [19, 20], coalesced and aligned memory access [39], or hardware-level schedules [32, 64]. 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 to our work.

### 2.4 Multi-GPU-based graph computing

There are some works that implement multi-GPU solutions for graph processing. Most graph processing frameworks focus on graph traversal algorithms: GUM [36] focuses on the workload imbalance in graph traversal problems, where graphs are transferred among GPUs dynamically using work stealing strategy; Digraph [62] proposes a path-based asynchronous execution model to accelerate iterative graph processing among multiple GPUs. It first partitions the original graphs into many subgraphs, then finds asynchronous paths in those subgraphs. Generally, graph partitioning is more friendly to traversal-based graph algorithms, because they have regular access patterns and the access to inter-partition edges (vertices) is limited in each iteration.

There are also some specific GPM implementations in multi-GPU platforms: PBE [22] is a subgraph matching implementation, which uses a graph partition approach (METIS [28]) to minimize the number of inter-partition edges. It introduces special techniques to deal with inter-partition matches. TRUST [41] is a triangle counting implementation, which partitions the original with a hash-based method to the source and destination of edges, and counts triangles in every two partitions on GPU. In conclusion, GPM has unpredictable access patterns; therefore, it is usually necessary to adopt an algorithm-specific graph partitioning method or execution design for multi-GPU implementation.

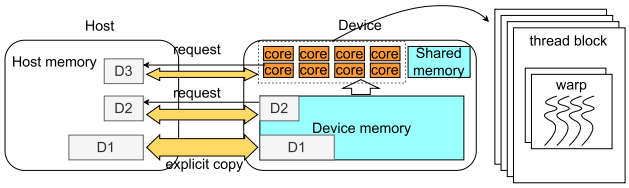



**Fig. 1** GPU architecture

## 3 Background

### 3.1 Heterogeneous system architecture

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

**Software.** A *warp* is a group of threads that 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.** A GPU has thousands of cores that share device memory. Shared memory is on-chip memory managed by thread blocks. It is limited in size (about 48 KB on V100 and 192 KB on A100 per thread block) but has low access latency. We use shared memory to optimize set intersection, an important primitive in GPM algorithms (see Sect. 6.1). Device memory is connected to host memory via PCIe. Data transfer between the host (CPU) and device (GPU) is a critical part of GPU-optimized algorithms. Most graph algorithms are memory-intensive, so the memory usage has a great influence on the overall performance. We briefly review the features of host memory and shared memory in Sect. 3.2 and 3.3, respectively.

### 3.2 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 [22, 26, 29, 41] 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 [45]. 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 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 the proper host memory access manner for different pages.

### 3.3 Shared memory usage

As noted earlier, shared memory is GPU on-chip memory, which is a programmable cache to accelerate frequent memory access. It has a much higher bandwidth than global memory; therefore, a reasonable strategy of using shared memory brings about great performance improvement to some key operations (such as set intersections) in GPM. This memory is shared by thousands of threads and divided into equally sized memory modules (called banks) that can be accessed simultaneously. Figure 2 shows the layout of banks in shared memory. Therefore, memory access to data in different banks can be served simultaneously, leading to high memory bandwidth. However, if multiple memory addresses of a memory request map to the same memory bank, the accesses are serialized, which is referred to as "bank conflict" [1]. Bank conflicts cause severe performance decline; therefore, it is desirable to reduce bank conflicts to maximize memory bandwidth.

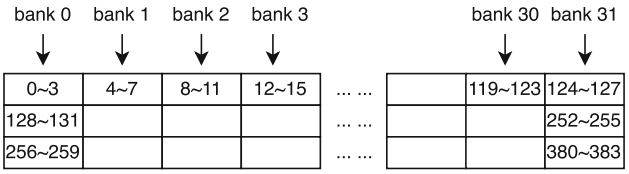



**Fig. 2** The memory banks in shared memory

Figure 2 demonstrates the banks of shared memory in a modern GPU architecture. Every 32-bits (4 bytes) belong to a bank, and there are 32 banks in total. Considering a consecutive memory space starting from byte 0: {byte 0–byte 3}, {byte 128–byte 131} belong to bank 0; {byte 4–byte 7}, {byte 132–byte 135} belong to bank 1. Data resident in different banks can be fetched in parallel, while data in the same bank can only be accessed serially. For example, byte 0 and byte 127 can be accessed in parallel, but byte 0 and byte 128 need to be accessed serially. We will discuss how to optimize shared memory access by reducing bank conflicts in GAMMA (Sect. 6).

## 4 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 $G_d$.

Figure 3 demonstrates GAMMA's system overview. We build a storage layer and an execution engine. The embeddings are arranged as an embedding table (Sect. 5.2). We use both host memory and device memory to store graph data and the embedding table (Sect. 5.1). In the execution engine, we provide three primitives to build various algorithms (Sect. 4.2), and adopt three optimizations to improve their performance (Sect. 6). We also propose optimizations for set intersection (Sect. 6), which is a key operator in GPM. The stars in Fig. 3 mark our main contributions. Table 2 lists all frequently-used notations throughout this paper.

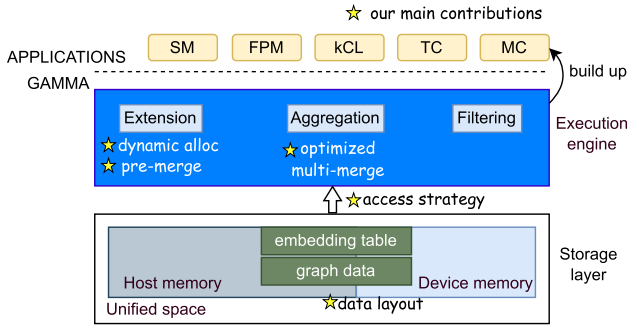



**Fig. 3** GAMMA system overview

**Table 2** Frequently used notations

| Variable | Description |
|---|---|
| *Graph attributes* | |
| $d_{max}$ | The maximum degree of vertices |
| $l(v)$ | Adjacency list of vertex v |
| $N_v$, $N_e$ | Neighbor vertices and neighbor edges |
| $M$ | Embeddings |
| $(u_1, u_2, u_3)$ | A vertex-oriented embedding instance |
| $(e_1, e_2, e_3)$ | A edge-oriented embedding instance |
| $ET$, $PT$ | Embedding table, pattern table |
| $v$-$ET$ | Vertex-oriented embedding table |
| $e$-$ET$ | Edge-oriented embedding table |
| $G_q$, $G_d$ | Query graph and data graph |
| *variables in memory access* | |
| $p$ | Physical memory page |
| $SpatialLoc$ | Spatial locality of a page |
| $temporalLoc$ | Temporal locality of a page |
| $AccHeat$ | Access heat of a page |
| $times(l(v))$ | The access time of l(v) |
| $A$ | All accessed adjacency lists |

## 4.1 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*).

Figure 4 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. 4b 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. 4b records the matched edges of $E_1$ in $G_q$.

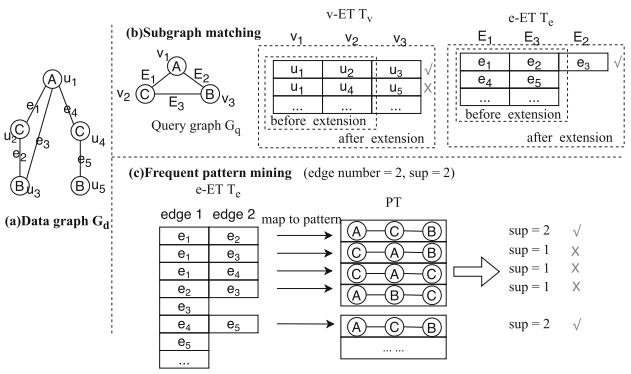



**Fig. 4** An example of edge-extension and vertex-extension in SM and FPM

## 4.2 Execution workflow

GAMMA has a three-phase execution process: "extension-aggregation-filtering" [12, 16, 63].

### 4.2.1 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 1** (Extension) Given an embedding $M$, the *vertex extension* ($Ext_v(M)$) and *edge extension* ($Ext_e(M)$) of $M$ are defined as follows:

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

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

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

where $N_v(u')$ and $N_e(u')$ 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. 4c. SM can use both types of extensions: edge extension can implement a *binary join* (query-edge-at-a-time) [30] and vertex extension can implement a *worst-case optimal join* (query-vertex-at-a-time) [38]. Figure 4b 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.

#### 4.2.2 Aggregation

This step maps an embedding table $ET$ into a pattern table $PT$ over which it computes an aggregation function. 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. 4c. This can be achieved by computing graph canonical label [60].[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.

#### 4.2.3 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. 4b, 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.

### 4.3 Implementing GPM tasks—examples

We illustrate the application of the described workflow by implementing SM and FPM. Figure 5 lists the data structures and interfaces visible to users in GAMMA.

#### 4.3.1 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.

[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.

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 Gd; //the data graph
5. query_graph Gq; //the query graph
```

Interfaces visible to users

```
1. Vertex_Extension (embedding_table ET, graph_data Gd);
2. Edge_Extension (embedding_table ET, graph_data Gd);
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. 5** GAMMA data structures and interfaces

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. 4b (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 δ_v denote the matching order of vertices in G_q;
2 ET ← all matched vertices to the first vertex in δ_v;
3 foreach unmatched vertex v ∈ δ_v do
4     Vertex_Extension(ET, G_d);
5     Filtering(ET, Constraint = G_q);
6 end
7 output_result(ET) ;
```

#### 4.3.2 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). Figure 4c 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 ← all length-1 embeddings, PT ← φ;
2 foreach i ∈ [1,l] do
3     PT = PT ∪ Aggregation(ET, mf);
4     Filtering(ET, PT, Constraint = sup_min);
5     if i<l then
6         Edge_Extension(ET, G_d);
7     end
8 end
9 output_result(PT);
```

## 5 GAMMA: system implementation

In this section, we discuss the implementation details of GAMMA. Specifically, we first discuss data storage and access of graph data and embedding tables in Sects. 5.1 and 5.2, respectively. Then, we extend GAMMA to multi-GPU architecture in Sect. 5.3.

### 5.1 Data graph data storage and access

We adopt Compressed Sparse Row (CSR) [12, 26, 59] to represent a data graph $G_d$, which is made up of adjacency lists of all vertices. GAMMA is an out-of-core system, since a large graph cannot be resident in GPU device memory. As mentioned earlier, GPM tasks always generate a large number of intermediate results, thus, a portion of device memory needs to be reserved for intermediate results to ensure high write performance. This reduces the space left for graph data in the device when processing large data graphs. Therefore, we maintain the data graph $G_d$ in host memory and propose a self-adaptive host memory access strategy. The maintenance of intermediate results (embedding tables) are discussed in Sect. 5.2.

As mentioned in Sect. 3.2, unified memory access [20] is friendly to data with good spatial or temporal locality, while zero-copy memory access [39] 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 10–15 GB, which is not a significant concern for host memory.

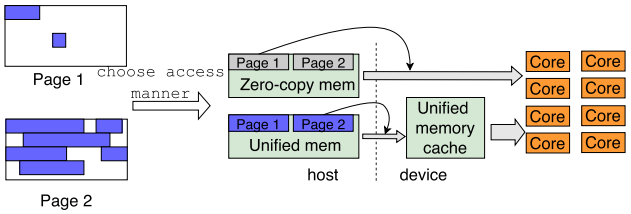



**Fig. 6** Two different host memory access methods

Adjacency lists are organized in memory pages. The key issue 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 can locate 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. 6), 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. 6).

#### 5.1.1 Spatial locality

Existing work [54] 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 2** (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.

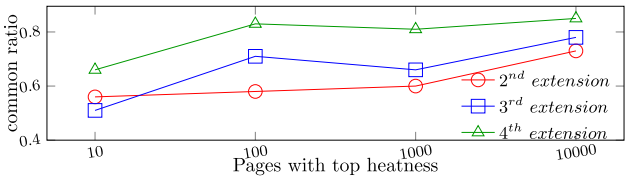



**Fig. 7** We take four extensions in SM in different datasets, and show the ratio of duplication of most frequently accessed pages between current extension and past extensions

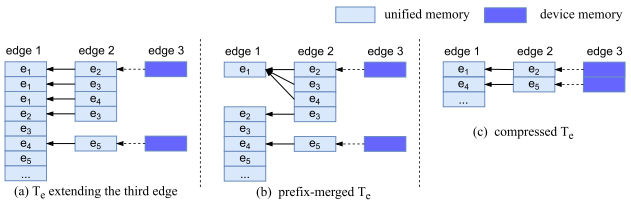



**Fig. 8** Data structure and data layout of the embedding table

#### 5.1.2 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. 7) 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 3** (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.,

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

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. 7). 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.

#### 5.1.3 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** (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.

#### 5.1.4 Access heat-based adaptive access model

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.

### 5.2 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 [5, 12]. For example, e-ET $T_e$ in FPM in Fig. 4c is extended to the third edge, as shown in Fig. 8a, b shows an embedding table after merging common prefixes.

Some embeddings are invalid after "filtering", and compressing the embedding table will save much space, as shown in Fig. 8c. The space compression also provides a better chance for coalesced memory access. However, the compression is ignored in existing GPM frameworks [12, 16, 50, 63]. 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. 8c) 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. 8, and flush them to host memory after the extension of embeddings.

### 5.3 Multi-GPU solution

Up to this point, we assume that GAMMA has only a single GPU. In order to properly scale, we extend GAMMA to a multi-GPU solution. Traditional multi-GPU GPM algorithms need to incorporate a specific graph partitioning strategy to place data across different GPU memories (as discussed in Sect. 2.4). Obviously, it is difficult to develop a generic graph partitioning solution for all GPM tasks. Therefore, we adopt the idea of "disaggregation of computation and storage" as has been adopted in recent cloud-native database systems in the multi-GPU version of GAMMA.

The storage layer of the multi-GPU architecture is the same as that of the single-GPU version. Specifically, GAMMA maintains the entire graph and the embedding tables in the host memory. Each GPU is an independent execution engine and works the same was as in the single-GPU environment, as illustrated in Fig. 9. Each GPU adopts the self-adaptive memory access strategy to address graph data and has its own cache for unified memory. The embedding table is divided into a number of task blocks, each containing roughly the same number of embeddings. During execution, each GPU is assigned an independent task block to process different embeddings. Although different task blocks may have different workloads due to different vertex degrees, this is a dynamic task assignment scheme. Once some GPU has finished its assigned task, it can fetch new tasks. Thus, the dynamic work distribution method assures good workload balance. Furthermore, our solution is agnostic to the specific graph partitioning strategy and has no communication among GPUs. Thus, it has a good scalability with respect to the number of GPUs.

During execution, both "extension" and "filter" primitives are performed on a single embedding; therefore, the execution of those two primitives is identical to those in single-GPU mode. The "aggregation" primitive needs to calculate some statistical information of all embeddings in the host memory. Our proposed optimization 3 in Sect. 6.2 helps to sort large numbers of embeddings, which is all performed

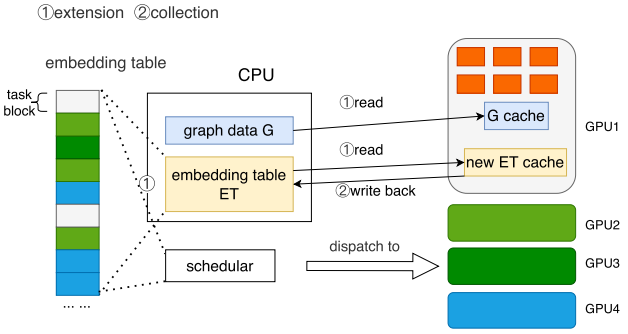



**Fig. 9** Multi-GPU solution of GAMMA

on GPUs. Therefore, all three primitives can be migrated to multi-GPU architecture.

## 6 GAMMA: optimization techniques

In this section, we discuss optimizations of GPU execution engine in GAMMA. As mentioned earlier, there are three primitives: extension-aggregation-filtering. Filtering is always done together with extension or aggregation. Therefore, we focus on optimizing *extension* and *aggregation*. Note that these proposed optimization techniques work for both single-GPU and multi-GPU architectures, since each GPU execution engine works independently.

### 6.1 Optimizing *extension*

**Challenge 1: Parallel Write Conflict.** When thousands of threads on a GPU are performing 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 [12], 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 [59] 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.

**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

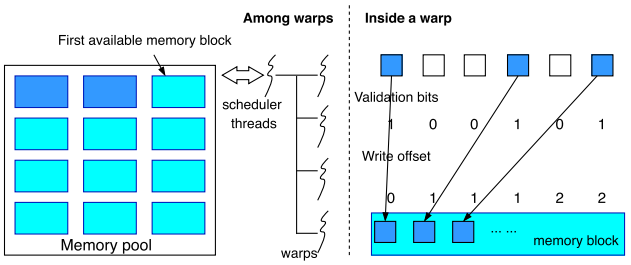



**Fig. 10** Dynamic memory allocation

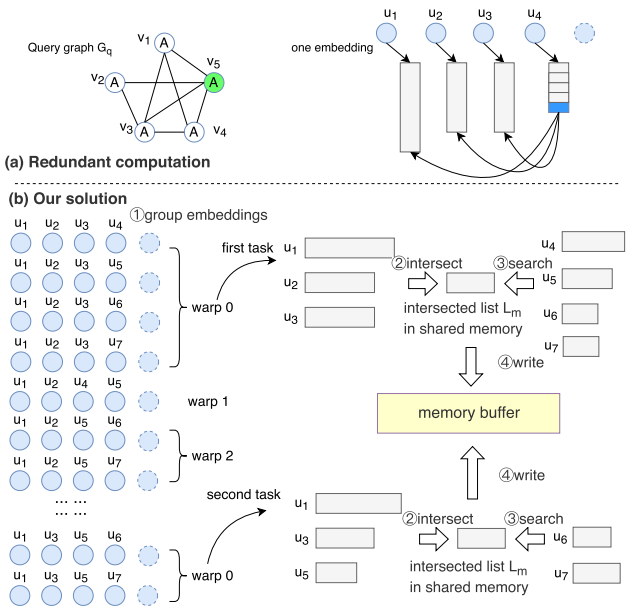



**Fig. 11** Redundant computation and our solution

shown in Fig. 10. 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 implementations on GPU, such as Pangolin [12], 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. 11a). 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. 11b: 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. 11b). 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. 11b. 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: Set Intersection.** More than half the running time is spent on set intersection in many GPM algorithms [23], e.g., k-clique, subgraph matching. Therefore, accelerating set intersection is vital to GAMMA. A number of

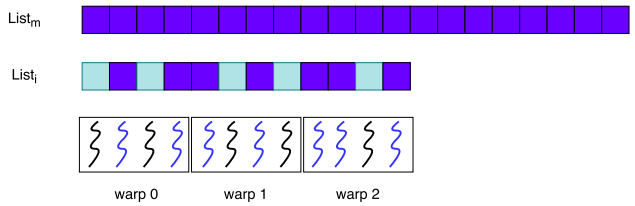



**Fig. 12** Using bloom filter for list intersection

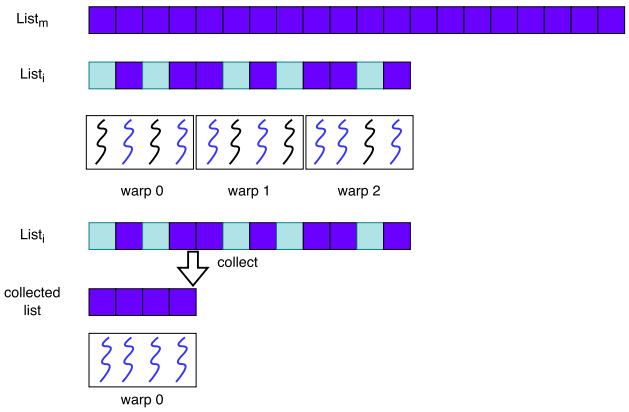



**Fig. 13** Three-stage process to avoid thread divergence in set intersection

alternative methods have been proposed for set intersections on GPU, including hash-based [40], bitmap-based [7] and liner algebra [56] methods. Most of these aim at GPU implementation of general set intersections, and they ignore specific features of intersections in GPM, resulting in unsatisfactory performance.

In GPM, one set often needs to be intersected with *multiple* sets. For example, in the subgraph match example in Fig. 11, the task assigned to the first warp (in Fig. 11b) requires list $L_m$ to be intersected with the adjacency lists of $u_4$, $u_5$, $u_6$ and $u_7$ to obtain different extended embeddings. In the triangle counting task, given a vertex $src$ with $n$ neighbors $dst_1, \ldots, dst_n$, the adjacency list of $src$ needs to be intersected with $n$ different adjacency lists to count all triangles including $src$. For this computation, an index can be built on the set that needs to be intersected multiple times. That is different from pairwise set intersection problem in GPU studied previously [7, 40, 56].

Furthermore, GPU architecture is different than CPU, which requires algorithmic adaptation with new challenges and opportunities.

**Optimization 3.** We propose two optimization techniques for set intersection.

**Three-Stage Set Intersection.** Assume one set ($L_m$) needs to be intersected with multiple sets $List_i, i = 1, \ldots n$. For example, in the example given in Fig. 11b, there would be $List_4$, $List_5$, $List_6$, abd $List_7$ corresponding to $u_4$, $u_5$, $u_6$ and $u_7$, respectively. In this case, building an auxiliary data structure on $L_m$ is worthwhile: the performance gains of multiple intersections make up for the overhead of building auxiliary data structures and bring extra benefits.

Bloom filter is a widely used method to accelerate checking existence of an element in a given list. In the case outlined above, building a bloom filter on $L_m$ would facilitate the intersection: each thread is responsible for searching an element $e$ in $List_i$ over the the bloom filter of $L_m$.

Obviously, bloom filter may lead to false positives, thus requiring a further check for each identified element $e$. Therefore, the naive implementation may lead to severe thread divergence. Since warps do SIMT execution, even if only a single thread needs to search over the list, all other threads in the same warp are idle until the search is completed. As shown in Fig. 12, only four blue elements pass the bloom filter, but all three warps are busy during the search process, which is a great waste of computation, especially when the size of intersection is far smaller than either of the original lists [23, 61].

To address the above problem, we propose a three-stage process, i.e., *filter-collect-search*, as shown in Fig. 13. In the *filter* stage, each thread checks the existence of an element of $List_i$ in the bloom filter of $L_m$, and marks if it passes the bloom filter check. All passed elements are collected into a continuous space in the *collect* stage. Finally, these collected elements are re-distributed to threads and each thread checks the existence of one element using binary search. Obviously, this avoids the thread divergence issue. Furthermore, the collected elements are much fewer than the original list; thus, the search needs much fewer threads than the filter stage.

Among existing systems that build auxiliary structures to accelerate list intersections, using hash tables is another popular choice [40, 41, 53]. However, bloom filters have two advantages in boosting intersection on GPUs. First, a bloom filter requires much less memory than a hash table built from the same list. For instance, for a set with 375 32-bit distinct integers, A bloom filter of an expected false positive probability of 0.05 takes up 292 Bytes. However, a hash table with a load factor of 0.9 requires 1667 Bytes, which is six times larger. In this case, a bloom filter can fit into shared memory but a hash table can not. Second, although we still need to access $L_m$(usually stored in shared memory) for final verification due to false positive queries, our proposed three-stage intersection design effectively triggers coalesced memory access. By contrast, searching in a hash table features random memory access because hash functions map different search targets to random positions.

**Memory-Friendly Data Layout.** The last step of the three-stage algorithm performs a *binary search* to check the existence of an element in the underlying list (e.g., $L_m$). Since it has higher bandwidth than global memory, shared memory is often used to speed up binary search as long

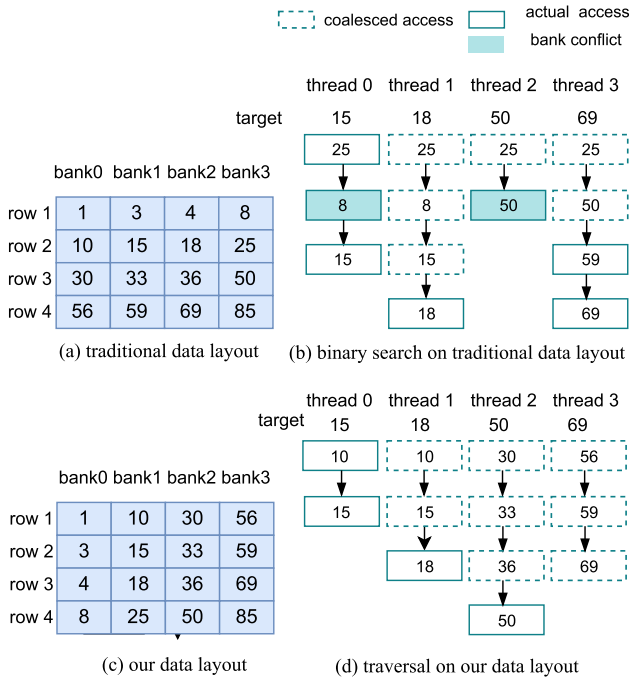



**Fig. 14** A memory-friendly data layout

as the shared memory can accommodate the list $L_m$. The degree distribution of real-world graphs usually follow the power-law; therefore, even in large graphs, most adjacency lists are sufficiently short to be resident in shared memory, although the space of shared memory for each warp is limited (in our setting, the shared memory space for each warp can accommodate about 375 32-bit integers, i.e, 375 neighbors). However, binary search has an unpredictable access pattern, which may cause *bank conflicts* (discussed in Sect. 3.3). To avoid this, we propose a new data layout in shared memory to boost the search performance over lists under the length of 375.

Suppose there are four threads in a warp and four banks in the shared memory; there is a non-descending list $L_m$ containing 16 elements in Fig. 14, where each element is a 32-bit integer. The default approach is to put all elements (in $L_m$) in row-first fashion (e.g., Fig. 14a), then each thread performs binary search to check one element. For example, four threads search four elements 15, 18, 50, and 69 in Fig. 14b, respectively. In the first round, all memory accesses coalesce, since all threads need to check the same middle value (25) due to binary search. In the second round, thread 0 and thread 1 need to access the first half. Thus, they access the same middle value (8) in the same transaction. The same holds for threads 2 and 3 for accessing the middle value (50) of the second half in another transaction. Although shared memory has enough bandwidth to parallelize multiple memory access transactions simultaneously, the visited values 8 and 50 are in the same bank. Due to bank conflict, the two transactions are executed sequentially. Generally, there are seven memory transactions and one conflict in the traditional row-first layout.

In the proposed column-first layout, searching for the same four elements requires only four memory transactions and incurs no bank conflict (Fig. 14d). We use a row-traversal method instead of binary search for searching elements. In the first round, all elements in the first row (1, 10, 30, 56) are accessed in a single transaction. Each thread can check to which column its target element belongs by conducting a binary search in the first row. For example, $10 \leq 15 < 30$, thus thread 0 only needs to check the second column(bank 1) in the subsequent rounds. Then the warp of four threads fetches the matrix row-by-row, during which each thread only needs to compare the target with one element in the specific column. In this way, we can guarantee that no bank conflict will happen because all threads access elements in different banks or the same element in one bank at each iteration. After processing all rows, each thread can tell whether its target element has a match in the list $L_m$. The process can be terminated early once all threads find their target elements. A warp is searched by accessing the list only once, and each memory access fetches a row in the matrix in Fig. 14c. In this way, we achieve a complexity of searching for a target of $O(log(B) + \lceil \frac{N}{B} \rceil)$, where $N$ represents the size of the list being searched and $B$ denotes the number of banks (or columns). Though this complexity is slightly higher than that of a binary search, avoiding bank conflicts can compensate for this gap and improve our performance, according to experimental results in Section 7.8.

## 6.2 Optimizing aggregation

**Challenge 4: 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 [21, 46], 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 4.** We first partition $PT$ into segments $S_i$ ($i = 1, \ldots, n$) such that each segment $S_i$ can be sorted by in-core GPU sorting algorithms [48]. 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. 15a, 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 $S_i$ ($i = 1, \ldots, n$) to get a set $\Omega$ (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** (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 $S_i^o$, $o = 0, \ldots, |\Omega|$ (see Fig. 15a). So we divide the task of merging $n$ segments $S_i$ ($i = 1, \ldots, n$) 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. Figure 15a gives an example. The first subtask merges all $S_i^0$ ($i = 1, \ldots, n$) (marked in blue), which are smaller than the first checkpoint $c_2$. Thus, the merged list of all $S_i^0$ ($i = 1, \ldots, n$) should precede the list merging all $S_i^1$ ($i = 1, \ldots, n$).

For explanation, we only discuss how to merge each 0-th list in all $S_i$ ($i = 1, \ldots, n$) (lines 7–23). Figure 15a 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 $S_2^0[i]$=$x$): we search the matched index of $x$ in all other lists (i.e., $S_1^0$ and $S_3^0$ in Fig. 15b), 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. 15b. We perform the same process for each element in all sorted lists $S_i^0$ ($i = 1, \ldots, n$) to locate elements in the merged list $S_m^0$, which can be parallelized.

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. 15c) 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 the 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 $S_3^0$. Figure 15c 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.

**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 ∈ 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 ∈ [1,n] do
11         foreach k ∈ [1,j) do
12             matched_idx[], matched_cnt[] ← zeros();
13             /* thread-wise searching for matches*/;
14             foreach p ∈ [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 ∈ S_1^i,...,S_n^i do
26         parallel_writing(S_j^i, writing_pos[j], global_off);
27     end
28 end
```

This process of sorting embeddings beyond the size of device memory is easy to extend to multi-GPU solutions. Firstly, all embeddings can be partitioned and sorted into multiple sorted segments, just as the input of Stage 1 in Fig. 15a. Then we perform the multi-merge process as discussed above. The processing of each ($S_j^i$, $S_k^i$) pair (lines 10–23 in Algorithm 3) can be dispatched to multiple GPUs, since they are independent tasks.

Sorting large numbers of embeddings is necessary for the "aggregation" primitive, and our proposed methods improve its performance for both single-GPU and multi-GPU solutions.

### 6.3 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 [12]. Here we give the worst-case complexity analysis.

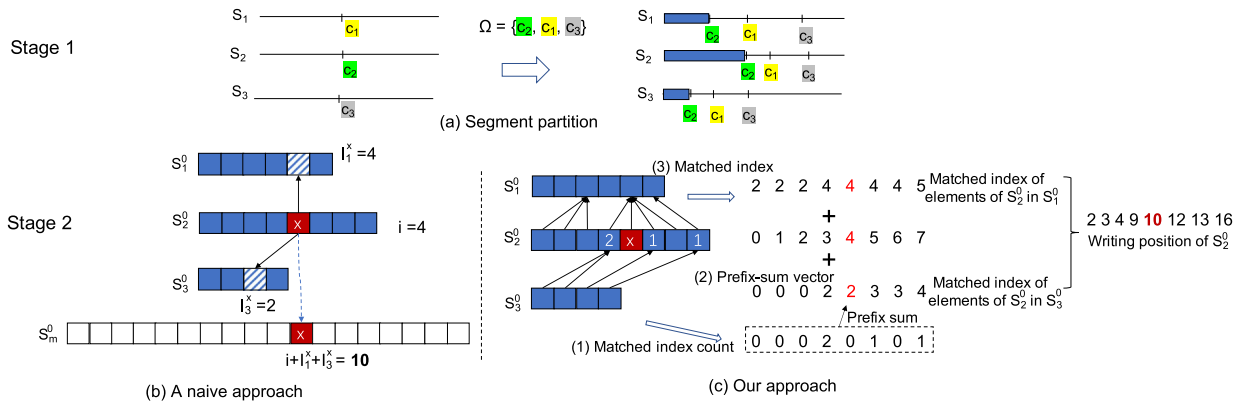



**Fig. 15** Two stages of multi-merge

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 [12].

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 the isomorphism test for each new embedding is $O(e^{\sqrt{klogk}})$ [6]. Considering combinatorial enumeration and isomorphism check, the complexity is $O(n^{k-2}(k-2)d_{max}+n^{k-1}d_{max}(log(d_{max})+e^{\sqrt{klogk}}))$ in the worst case. Other GPM algorithms can be analyzed similarly.

Assuming there are $w$ warps in the device, the 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.

**Table 3** Datasets Infos

| Dataset | Nodes | Edges | Types |
|---|---|---|---|
| cit-Patent(CP) | 6 M | 17 M | Citation |
| com-lj(CL) | 4 M | 34 M | Social |
| com-orkut(CO) | 3 M | 117 M | Social |
| email-Euall(EA) | 265K | 729K | Email |
| email-Enron(ER) | 37K | 368K | Email |
| gowalla(GW) | 197K | 2 M | 2,273,138 |
| com-lj×8(CL×8) | 32 M | 467 M | Synthetic |
| soc-Live×5(SL×5) | 24 M | 481 M | Synthetic |
| uk2005(UK5) | 39 M | 1.6B | Web |
| it2004(IT) | 41 M | 2.1B | Web |
| twitter_rv(TW) | 62 M | 2.4B | Social |
| uk2014(UK14) | 196 M | 14.8B | Web |

## 7 Experiments

### 7.1 Experimental setting

**Infrastructure.** We use the CUDA−9.0 toolkit and GCC 5.3.0 to compile all codes with -O3 option. All experiments on a single GPU are carried out on a Linux server with a 32-core Intel Xeon E5-2640 CPU and 380 GB of host memory. It also has an NVIDIA Tesla V100 with 16 GB of global memory. For multi-GPU solutions, we test them on a cloud server with 8 V100 GPUs.

**Datasets.** We use several real graphs with varying sizes from different domains. To test the scalability of GAMMA to large graphs, we scale up *soc-Live* and *com-lj* by 5× and 8× using graph upscaling technique [42]; we also use

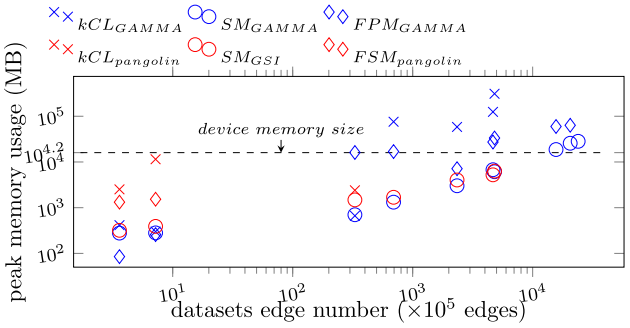



**Fig. 16** Peak memory usage

billion-scale real-world graphs. Table 3 lists all datasets. We also include uk2014 data set with more than ten billion edges for the scalability test. Datasets of such scale are never tested in previous work. For the evaluation convenience, we sampled 30% of its edges because existing systems cannot run SM, kCL, and FPM on the entire uk2014 graph and end in a reasonable time.

**Comparative evaluation.** We use subgraph matching (SM), frequent pattern mining (FPM), and k-clique (kCL) workloads to compare GAMMA[2] with baselines (Pangoline [12], Peregrine [27], GSI [59], and GraphMiner [9]) and the state-of-the-art GPU-based solutions (PBE [22], G$^2$Miner [10], and GraphSet [47]).

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 [59], 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 the FPM implementation in GraphMiner [9], which is a parallel graph algorithm library that combines several state-of-the-art GPM designs [11–13].

## 7.2 Memory usage

The peak memory usage of GAMMA and other GPU-based GPM implementations, including the host memory and device memory, is shown in Fig. 16. GAMMA uses less memory than other GPM implementations for a given input graph, thanks to our compression of the embedding table. There are many cases in GAMMA where memory usage exceeds available device memory. The maximum memory consumption reaches 310 GB in some large graphs. In-core GPM algorithms only use device memory and cannot run on large graphs, while GAMMA is the first to support out-of-core GPM on GPU by adaptively storing the data graph in the host memory.

Figure 16 also indicates that in general, SM consumes less memory than FPM, and FPM requires less memory than kCL when the edge number goes beyond $10^7$. In other words, the maximum graph size in SM is the largest among the three algorithms in our experiments, and that of kCL is the smallest. This is because SM has the most pruning conditions, while kCL has the fewest. We can also notice that the gap between these three algorithms is unstable. For example, when the edge size is around $10^7$, the memory usage of SM and kCL are close. When the edge size exceeds $10^8$, the gap increases, since kCL's intermediate result size increases rapidly, and therefore it is more likely to run out of GPU memory.

## 7.3 Comparative evaluation

GAMMA's comparative evaluation with the baselines on different workloads is discussed below.

**K-clique.** The experimental results of GAMMA compared with state-of-the-art works for kCL are shown in Fig. 19. "Pangolin-ST" denotes the single-thread version of Pangolin, and "Pangolin-GPU" denotes the GPU version. Some baselines crash on large graphs, so we omit those cases in the figure, but GAMMA shows good scalability on all datasets. Furthermore, GAMMA performs better than "Pangolin-GPU" and Peregrine, achieving an average speedup of 2.39× and 3.17×, respectively. Comparison with SOTA solutions indicates GAMMA's capability of settling the out-of-core problem and maintaining competitive performance. PBE is even slower than baseline systems in k-clique tests because many cross-partition queries are introduced by clique-like patterns. GAMMA beats PBE by 8.91× on average. G$^2$Miner also partitions large data graphs, so cross-partition edges can harm its performance. As a result, we can see that GAMMA is 13% faster than G$^2$Miner on large graphs like UK5.

**SM.** Fig. 17 reports the running time of GAMMA, GSI, and Peregrine for three SM queries in Fig. 21 on each dataset. Pangolin and GraphMiner are not included in the comparison since they do not implement subgraph-matching algorithm.

GAMMA performs much better than GSI and Peregrine on all large graph datasets except for two small datasets ($EA$ and $ER$), achieving 90.1% speedup over GSI and 173.9% speedup over Peregrine on all datasets. Preparing host memory storage in GAMMA accounts for much of the total running time for small datasets. Thus, it is slower than GSI with an in-core GPU implementation and the CPU-based Peregrine. GSI and Peregrine crash on some datasets in our experiments, which we omit in Fig. 17. The reason for these crashes is the enormous size of the intermediate results. As shown in Fig. 17, GAMMA manages to finish all queries with the proposed compressed storage for the intermediate

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

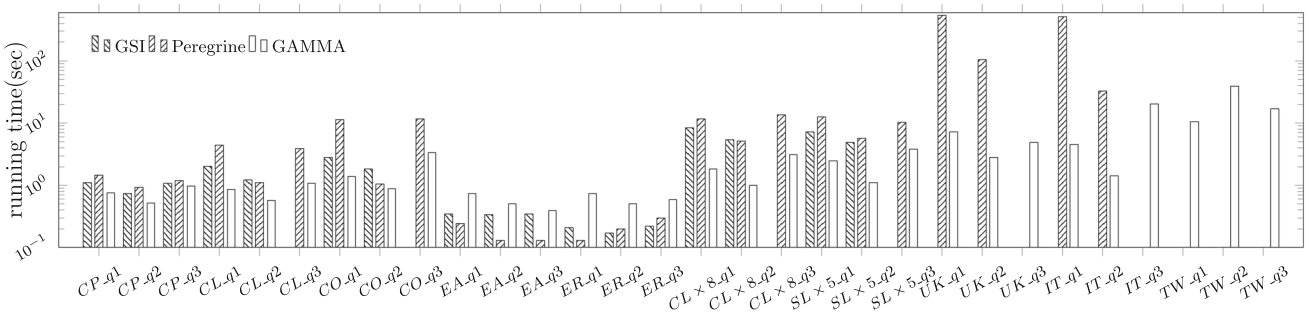



**Fig. 17** Performance of SM against baselines

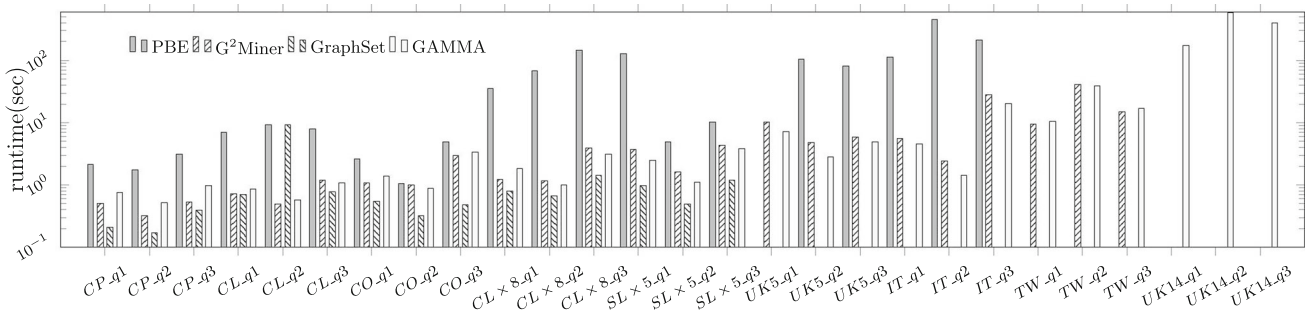



**Fig. 18** Performance of SM against SOTA solutions

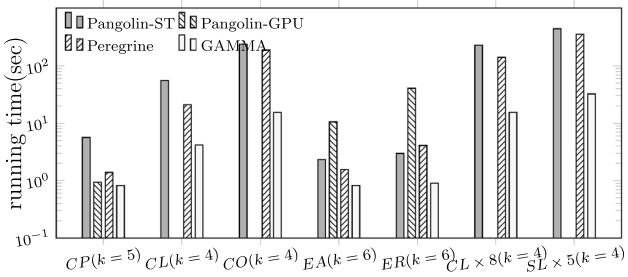



**Fig. 19** Performance of kCL against baselines

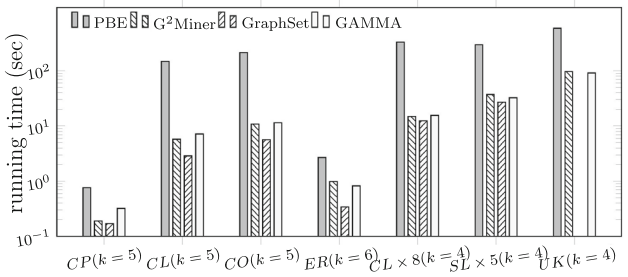



**Fig. 20** Performance of kCL against SOTA solutions

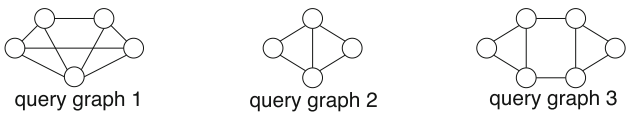



**Fig. 21** Query graphs in SM

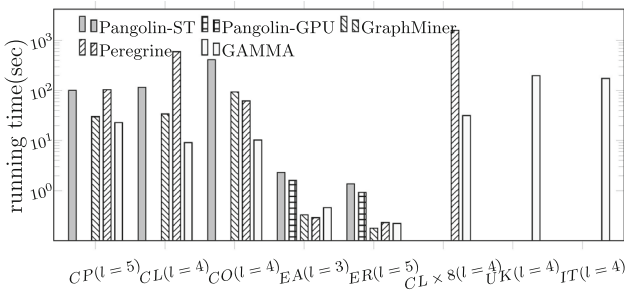



**Fig. 22** Performance of FPM compared with baselines

results. Although PBE narrows its performance gap with other systems, GAMMA still has an advantage of 4.51×. The comparison with $G^2$Miner has similar results with k-clique evaluation, where GAMMA and $G^2$Miner is equally matched on smaller graphs but GAMMA slightly prevails on larger graphs, thanks to GAMMA's unique memory management strategy.

**FPM.** We compare GAMMA with GraphMiner, Peregrine, and Pangolin in FPM. As shown in Fig. 22, GAMMA exhibits significant scalability advantages over other works. Specifically, GAMMA can process billion-scale graphs (UK14), while other methods meet crashes in these datasets. GAMMA has 86.1% and 73.8% performance improvement compared with "Pangolin-ST" and "Pangolin-GPU", respectively. Also, GAMMA achieves an average of 50.6% speedup compared with Peregrine. Although GraphMiner implements a specific FPM algorithm, GAMMA still prevails, achieving

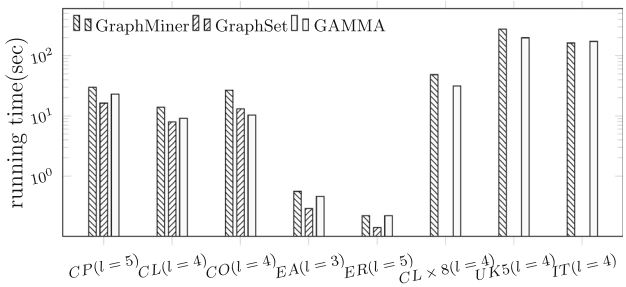



**Fig. 23** Performance of FPM compared with SOTA solutions

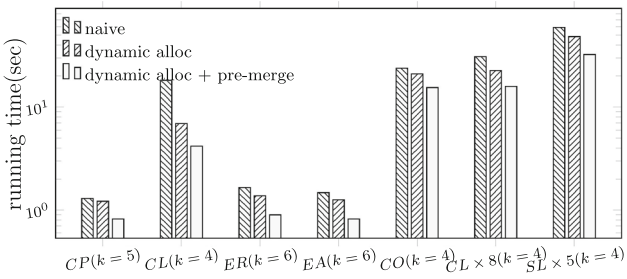
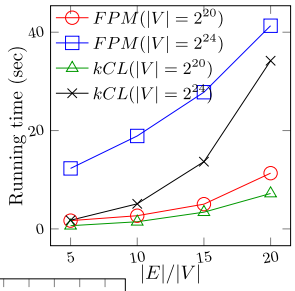



**Fig. 26** The effect of our optimizations on kCL

**Fig. 24** Different graph densities



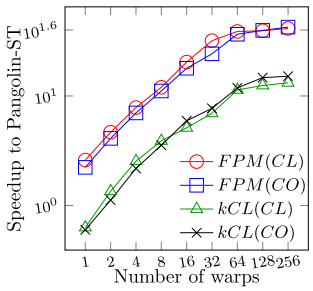




**Fig. 25** Different number of warps

24.7% performance improvements. GAMMA shows performance superiority due to the optimization of the *aggregation* primitive (Optimization 3 in Sect. 6). This alleviates the device memory limit and allows the aggregation of huge embedding tables. Compared with baselines, GAMMA's optimized three-phase processing framework guarantees performance superiority in FPM. It is worth mentioning that GraphSet is the fastest system in our evaluation, but it cannot deal with graphs larger than GPU memory. Therefore, it fail to process large graphs with more than 1B edges. Besides, the performance advantage of GraphSet becomes much smaller in the evaluation of frequent pattern matching, since GraphSet expects algorithms to contain perfect loops (i.e., nested loops with all computation only in the innermost loop) as much as possible, which is not the case for algorithms requiring aggregations like frequent pattern mining.

## 7.4 Scalability

We have evaluated GAMMA's scalability ranging from small to billion-scale graphs in Sect. 7.3. As shown in Figs. 17 and 22, 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 [31] with different numbers of vertices and graph densities. As shown in Fig. 24, GAMMA has good scalability for 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. 25, where we use the performance of "Pangolin-ST" as a baseline and plot GAMMA's normalized speedup. It outperforms "Pangolin-ST" with one warp or two warps and has approximately linear performance improvements as the warp number increases.

## 7.5 Evaluation of primitive optimizations

In this subsection, we evaluate the effectiveness of the three optimizations discussed in Sect. 6. The first two optimizations are related to the "extension" primitive. We design a dynamic memory allocation strategy, denoted as "dynamic-alloc" in the following figures. We also avoid duplicate computation by grouping embeddings with the same prefix. This optimization is marked as "pre-merge" in the figures. As a baseline, the "naive" method does not have either optimization. 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 Figs. 27 and 26, respectively.

Figures 27 and 26 show that both "dynamic-alloc" and "pre-merge" significantly improve performance, 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.

Sorting a list that exceeds the device memory is an essential operation for our aggregation primitive, and Optimization 3 reduces the computation time of this operation (called "multimerge+opt"). Existing works of out-of-core GPU sort usually involve considerable CPU processing [21, 46]. Thus,

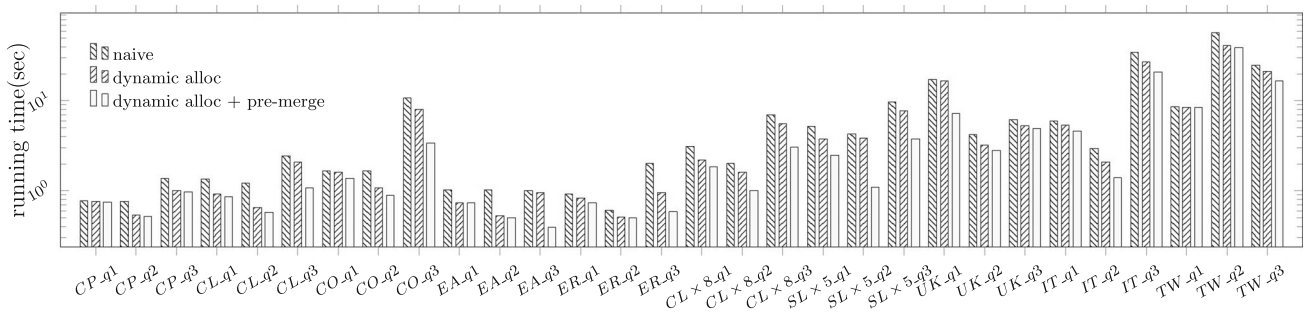



**Fig. 27** The effect of optimizations on SM

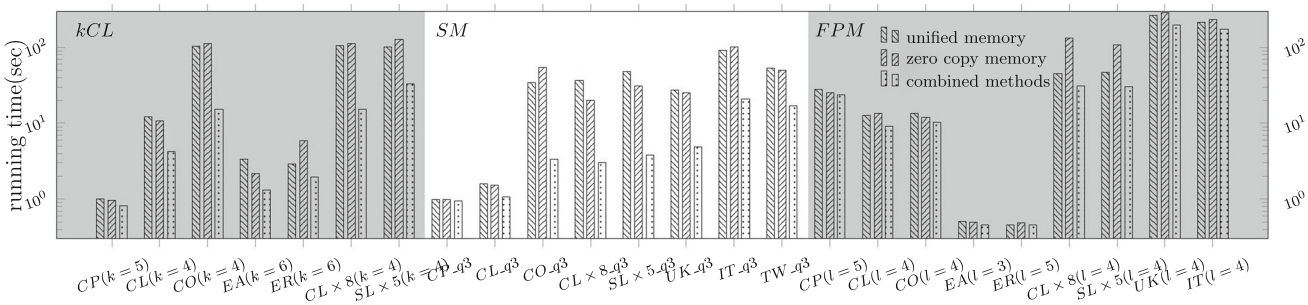



**Fig. 28** The effect of our memory access mode

**Table 4** FPM performance of different sorting methods.(sec)

| Dataset | 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 |

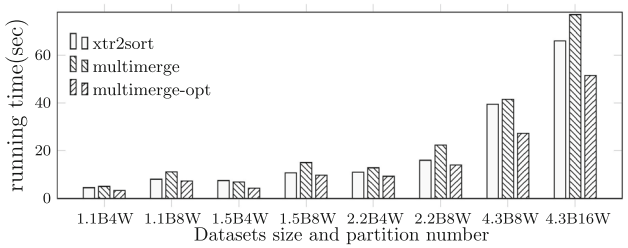



**Fig. 29** Effect of Optimization 3 on multi-merge

they cannot achieve maximum parallelism. We compare "multimerge+opt" with "cpusort", a popular sorting method implemented by Thrust [2] on CPU, and "xtr2sort" [46], 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" that searches each item of segments over other segments (see Fig. 15b). Not all datasets need the external sort of the pattern table PT, and we show the performance of two datasets as examples in Table 4, in which our approach runs the fastest compared with different baselines.

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 4, and we do not plot its results. In Fig. 29, horizontal axis labels denote tasks. For example, "4.3B8W" indicates that 8-way multi-merge is performed on 4.3 billion 64-bit values. From Fig. 29, we conclude that this optimization achieves 34.2% speedup over the naive implementation and 20.9% speedup over xtr2sort.

## 7.6 Evaluation of hybrid memory access

We evaluate GAMMA's memory access determination strategy over all three GPM workloads. The results are shown in Fig. 28. 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 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.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.

Figure 30 demonstrates time breakdown on computation, on-device memory access, and cross-device data communication to investigate our system bottleneck. We use a representative data graph, orkut, and report time breakdown

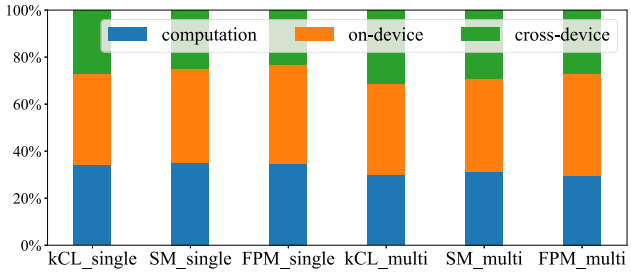



**Fig. 30** Running time breakdown of computation and communication on Orkut

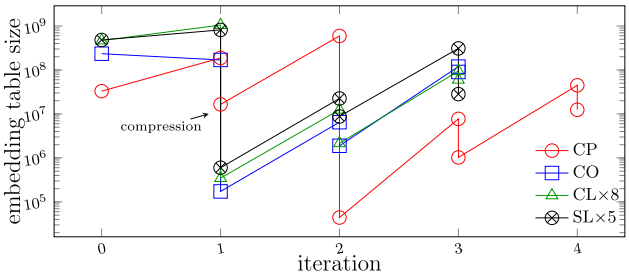



**Fig. 31** The necessity of compression of embedding tables

when running kCL, SM, and FPM with single-GPU and multi-GPU settings. Cross-device data communication takes up the lowest percentage (27.1% on average) thanks to our self-adaptive memory access strategy, which consider both spatial and temporal locality and dynamically adjusts memory access mode. Apart from verifying the impact of our hybrid memory access in reducing cross-device communication, we have two observations by analyzing this time breakdown. First, on-device memory access (i.e., accessing GPU memory) consumes the largest portion of running time. This is because graph pattern mining algorithms require reading neighbor lists repeatedly and writing back intermediate results to GPU memory. Second, we also notice that cross-device transfer is more time-consuming at the first few iterations and in the first run[3] because cross-device data transfer is inevitable when you request data for the first time.

These observations inspire us to possible future optimization. Although the size of intermediate results has been reduced, it still causes a large burden compared to computational costs, which is a disadvantage of the BFS-based computation manner. Therefore, DFS-based computation logic that produces much less intermediate results should be combined with our current system. Besides, since cold-start is a problem for implicit cross-device communication, a warm-up process or pre-load techniques may be worth considering.

[3] We report average performance of five runs.

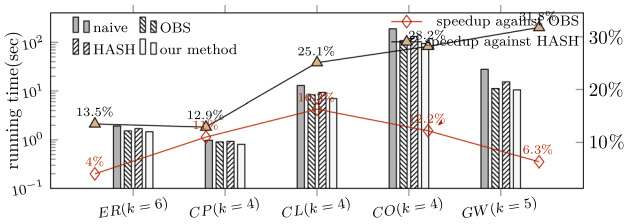



**Fig. 32** The performance of different intersection methods in k-clique

## 7.7 Compression of embedding table

In Sect. 5.2, we propose compressing the embedding table, which is not used in other GPM frameworks. In this set of experiments, we show the necessity of compression. Figure 31 shows the variation of the embedding table size for four datasets in FPM, which benefits most from embedding table compression. After each extension, the embedding table is compressed for the following three points: reducing memory usage and unnecessary enumerations of invalid embeddings in later extensions and achieving more coalesced memory access. As shown in Fig. 31, embedding table size shrinks sharply after compression in each iteration for all datasets. This demonstrates the necessity of the compression of embedding tables. Compression reduces excessive and unnecessary memory usage of the embedding table and plays a key role in GAMMA's ability to process large graphs.

## 7.8 Evaluation of set intersection methods and optimization

There are many general methods in the literature for set intersection on GPU based on hash method [40], binary search [26], bitmap [7] and static index [43]. However, GPM algorithms need a large number of parallel set intersections, which makes some methods unsuitable. For example, set intersection based on bitmap [7] and static index [43] need extra space, which causes much more space usage in GPM. Thus, these two methods are not suitable. In our experiments, we use set intersection implementation based on hash method [41] and optimized binary search (OBS) [26] as baselines to evaluate the effect of our proposed set intersection method.

Hash-based method builds a hash table on the short list of the two intersected lists, then each element in the long list searches in the hash table. The OBS-based method caches the search tree's top levels in shared memory to improve memory access performance. These two methods achieve good performance improvements on set intersections; however, they are general-purpose solutions, and our proposed set intersection method achieves better performance in GPM because of more algorithm-concerned optimizations.

Frequent pattern mining algorithms do not need set intersections. Thus, we evaluate the effect of different set

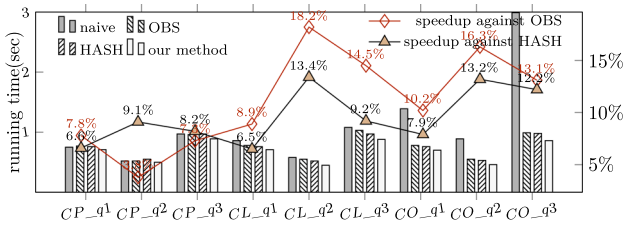



**Fig. 33** The performance of different intersection methods in subgraph matching

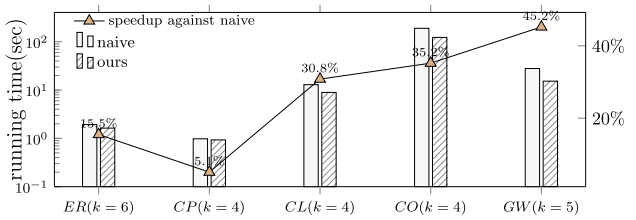



**Fig. 34** The effect of avoiding bank conflicts in k-clique counting

intersection methods in kCL and SM. We record all the required intersections throughout the execution and compare the running time of conducting these intersections with various methods instead of showing the running time of the entire algorithm. In this way, we can omit the influence of the other procedures in kCL and SM.

In Figs. 32 and 33, the "naive" method indicates the original method used in [25] without extra set intersection optimizations; "our method" denotes our intersection strategy with optimizations 2 and 3 proposed in Sect. 6. As shown in the figures, our method has better performance than the hash-based method and OBS-based method; it achieves 1.2× ∼ 2.65× performance improvement over the naive method in k-clique counting problem and achieves 1.04× ∼ 3.4× performance improvement in subgraph matching. What's worth mentioning is that the speedup is higher for kCL than SM because kCL requires intersections involving more neighbor lists, and the 3-stage intersection method achieves higher improvement when more sets are to be intersected.

We also evaluate the effect of our optimized memory layout in Sect. 6.1 in Fig. 34 to figure out their contribution to the overall performance. In the kCL algorithm, our method has an average of 1.19× speed up over the naive implementation. The performance gain is more obvious on larger datasets. In conclusion, our proposed optimizations for set intersections work well and improve performance compared to existing solutions.

## 7.9 Evaluation of multi-GPU performance

In this subsection, we conduct two experiments to evaluate the effect of our proposed multi-GPU implementation of GAMMA. We use kCL and FPM as examples for the convenience of presentation because the operations of kCL and SM are similar. First, we test the scalability of our multi-GPU implementation with an 8-GPU cloud server and report the running time using 1, 2, 4, and 8 GPUs. On the cloud server, the single-GPU implementation used as the baseline is slightly slower, but it's acceptable because we only examine the performance gain when the number of GPUs increases. Second, we compare GAMMA's multi-GPU implementation with existing systems: PBE [22], G$^2$Miner [10], and GraphSet [47]. We omit the smaller datasets, ER, CP, and EA because these datasets are too small for multi-GPU solutions to show performance gains. Their running time of actual computation is very short in a single GPU, so the overhead of setting up the environment takes up too much time in the multi-GPU version. There is no necessity to use multi-GPU implementation for small datasets.

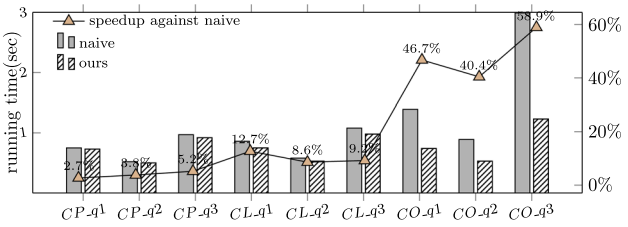



**Fig. 35** The scalability of multi-GPU implementation in kCL

Figure 35 shows the running time of kCL on different datasets with different numbers of GPUs. We set $k = 4$ by default. Our implementation achieves a 1.61× speedup with 2 GPUs, a 2.62× speedup with 4 GPUs, and 3.27× with 8 GPUs. When we use 8 GPUs, the speedup averaged by the GPU number is only 0.4, which is lower than our 2-GPU (0.8) and 4-GPU (0.65) solutions. However, if we compare the speedup for different datasets, we can see that our 8-GPU implementation achieves higher speedup on larger datasets (e.g., 4.15× on UK versus 2.36× on CL). When the dataset is relatively small and only a small workload is assigned to each GPU, the overhead from setup and scheduling dominates the performance. Our multi-GPU solution can scale to and suit billion-scale datasets better because we use a flexible strategy for data storage and a dynamic task assignment scheme, which reduces cross-device communication and workload imbalance.

FPM is more challenging when running in parallel because aggregation operation is required to gather the results from different GPUs to calculate the global pattern counts. However, as shown in Fig. 36, GAMMA still achieves a speedup of 1.44×, 2.34×, and 3.09× using 2,4 and 8 GPUs, respectively. The performance improvement is attributed to extending the sorting optimization to the multi-GPU (presented in Sect. 6). This optimization facilitates GAMMA to organize results from multiple GPUs efficiently.

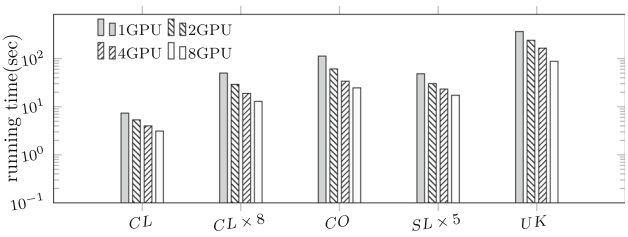



**Fig. 36** The scalability of multi-GPU implementation in FPM

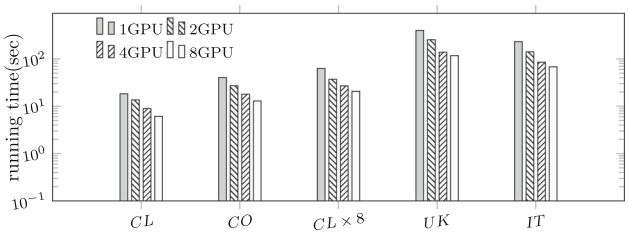



**Fig. 37** k-clique performance of multi-GPU GAMMA against SOTA solutions

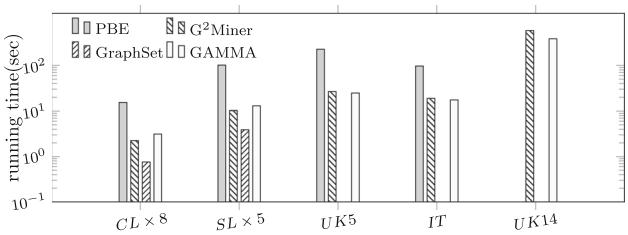



**Fig. 38** Frequent pattern mining performance of multi-GPU GAMMA against SOTA solutions

Compared to recent multi-GPU solutions, GAMMA maintains its advantage in terms of performance and scalability, according to Figs. 37 and 38. As mentioned in Sect. 7.3, although GraphSet achieves high efficiency on data graphs that can fit into GPU memory, it cannot handle the out-of-core problem in its multi-GPU implementation. Strategies to partition data graphs larger than GPU memory capacity are not the focus of GraphSet design. In this case, increasing the number of GPUs cannot enable GraphSet to support larger graphs effectively.

## 8 Conclusions

In this paper, we design GAMMA, an out-of-core GPM (graph pattern mining) framework on GPU, 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 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 challenges, and we give three optimizations to improve performance. Based on the design for single-GPU implementation, we scale GAMMA to a multi-GPU environment. Extensive experiments show that GAMMA outperforms state-of-the-art GPM frameworks and some dedicated graph algorithms on GPU.

**Acknowledgements** Lei Zou was supported by The National Key Research and Development Program of China under grant 2023YFB45-02303 and NSFC under grant 61932001 and U20A20174. Tame Özsu's research was funded by a Discovery Grant from Natural Sciences and Engineering Research Council (NSERC) of Canada. Lei Zou is the corresponding author of this work.

## References

1. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html?highlight=bank
2. https://thrust.github.io
3. Abdelhamid, E., Abdelaziz, I., Kalnis, P., Khayyat, Z., Jamour, F.: 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), pp 716–727 (2016)
4. Almasri, M., Hajj, I.E., Nagi, R., Xiong, J., Hwu, W.m.: Parallel k-clique counting on gpus. In: Proceedings of the 36th ACM International Conference on Supercomputing (ICS), pp 1–14 (2022)
5. Aoe, J.I., Morimoto, K., Sato, T.: An efficient implementation of trie structures. Softw. Pract. Exp. **22**(9), 695–721 (1992)
6. Babai, L., Kantor, W.M., Luks, E.M.: Computational complexity and the classification of finite simple groups. In: Proceeding of the Annual Symposium on Foundations of Computer Science (SFCS), pp 162–171 (1983)
7. Bisson, M., Fatica, M.: High performance exact triangle counting on gpus. IEEE Trans. Parallel Distrib. Syst. **28**(12), 3501–3510 (2017)
8. Chen, H., Liu, M., Zhao, Y., Yan, X., Yan, D., Cheng, J.: G-miner: an efficient task-oriented graph mining system. In: Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys), pp 1–12 (2018)
9. Chen, X.: Graphminer. https://github.com/chenxuhao/GraphMiner
10. Chen, X., Csail, M., Csail, A.M.: Efficient and scalable graph pattern mining on GPUs. arXiv:2112.09761 (2021). https://api.semanticscholar.org/CorpusID:245334945
11. Chen, X., Dathathri, R., Gill, G., Hoang, L., Pingali, K.: Sandslash: a two-level framework for efficient graph pattern mining. In: Proceedings of the ACM International Conference on Supercomputing (ICS), pp 378–391 (2021)
12. Chen, X., Dathathri, R., Gill, G., Pingali, K.: Pangolin: an efficient and flexible graph mining system on CPU and GPU. Proc. VLDB Endowment (PVLDB) **13**(8), 1190–1205 (2020)
13. Chen, X., Huang, T., Xu, S., Bourgeat, T., Chung, C., Arvind, A.: Flexminer: a pattern-aware accelerator for graph pattern mining. In: Proceeding of the Annual International Symposium on Computer Architecture (ISCA), pp 581–594 (2021)
14. Chu, W.T., Tsai, M.H.: Visual pattern discovery for architecture image classification and product image search. In: Proceedings of the 2nd ACM International Conference on Multimedia Retrieval, pp 1–8 (2012)

15. Deshpande, M., Kuramochi, M., Wale, N., Karypis, G.: Frequent substructure-based approaches for classifying chemical compounds. IEEE Trans. Knowl. Data Eng.(TKDE) **17**(8), 1036–1050 (2005)
16. Dias, V., Teixeira, C.H., Guedes, D., Meira, W., Parthasarathy, S.: Fractal: A general-purpose graph pattern mining system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp 1357–1374 (2019)
17. Eberle, W., Graves, J., Holder, L.: Insider threat detection using a graph-based approach. J. Appl. Secur. Res. **6**(1), 32–81 (2010)
18. Elseidy, M., Abdelhamid, E., Skiadopoulos, S., Kalnis, P.: Grami: frequent subgraph and pattern mining in a single large graph. Proc. VLDB Endow.(PVLDB) **7**(7), 517–528 (2014)
19. Gera, P.: Overcoming memory capacity constraints for large graph applications on gpus. Ph.D. thesis, Georgia Institute of Technology (2021)
20. Gera, P., Kim, H., Sao, P., Kim, H., Bader, D.: Traversing large graphs on gpus with unified memory. Proc. VLDB Endow.(PVLDB) **13**(7), 1119–1133 (2020)
21. Gowanlock, M., Karsin, B.: Sorting large datasets with heterogeneous cpu/gpu architectures. In: Proceeding of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 560–569 (2018)
22. Guo, W., Li, Y., Sha, M., He, B., Xiao, X., Tan, K.L.: Gpu-accelerated subgraph enumeration on partitioned graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp 1067–1082 (2020)
23. Han, S., Zou, L., Yu, J.X.: Speeding up set intersections in graph algorithms using simd instructions. In: Proceedings of the 2018 International Conference on Management of Data, pp 1587–1602 (2018)
24. Hu, L., Guan, N., Zou, L.: Triangle counting on gpu using fine-grained task distribution. In: Proceeding of the International Conference on Data Engineering Workshops (ICDEW), pp 225–232 (2019)
25. Hu, L., Zou, L., Özsu, M.T.: GAMMA: A graph pattern mining framework for large graphs on GPU. In: 39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023, pp 273–286. IEEE (2023)
26. Hu, Y., Liu, H., Huang, H.H.: Tricore: Parallel triangle counting on gpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp 171–182 (2018)
27. Jamshidi, K., Mahadasa, R., Vora, K.: Peregrine: a pattern-aware graph mining system. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp 1–16 (2020)
28. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
29. Kim, M.S., An, K., Park, H., Seo, H., Kim, J.: 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), pp 447–461 (2016)
30. Lai, L., Qin, L., Lin, X., Zhang, Y., Chang, L., Yang, S.: Scalable distributed subgraph enumeration. Proc. VLDB Endow. (PVLDB) **10**(3), 217–228 (2016)
31. Leskovec, J., Chakrabarti, D., Kleinberg, J.M., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: an approach to modeling networks. J. Mach. Learn. Res. **11**, 985–1042 (2010)
32. Li, C., Ausavarungnirun, R., Rossbach, C.J., Zhang, Y., Mutlu, O., Guo, Y., Yang, J.: 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), pp 49–63 (2019)
33. Lu, S., He, B., Li, Y., Fu, H.: Accelerating exact constrained shortest paths on gpus. Proc. VLDB Endow. **14**(4), 547–559 (2020)
34. Lü, Y., Guo, H., Huang, L., Yu, Q., Shen, L., Xiao, N., Wang, Z.: (2021) Graphpeg: Accelerating graph processing on gpus. ACM Trans. Archit. Code Optim. **18**(3), 30:1–30
35. Mawhirter, D., Wu, B.: Automine: harmonizing high-level abstraction and high performance for graph mining. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), pp 509–523 (2019)
36. Meng, K., Geng, L., Li, X., Tao, Q., Yu, W., Zhou, J.: Efficient multi-gpu graph processing with remote work stealing
37. Merrill, D., Garland, M., Grimshaw, A.S.: High-performance and scalable GPU graph traversal. ACM Trans. Parallel Comput. **1**(2), 14:1–14:30 (2015)
38. Mhedhbi, A., Kankanamge, C., Salihoglu, S.: Optimizing one-time and continuous subgraph queries using worst-case optimal joins. ACM Trans. Database Syst. (TODS) **46**(2), 1–45 (2021)
39. Min, S.W., Mailthody, V.S., Qureshi, Z., Xiong, J., Ebrahimi, E., Hwu, W.m.: Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. Proceedings of the VLDB Endowment (PVLDB) **14**(2), 114–127 (2020)
40. Pandey, S., Li, X.S., Buluc, A., Xu, J., Liu, H.: H-index: Hash-indexing for parallel triangle counting on gpus. In: 2019 IEEE high performance extreme computing conference (HPEC), pp 1–7. IEEE (2019)
41. Pandey, S., Wang, Z., Zhong, S., Tian, C., Zheng, B., Li, X., Li, L., Hoisie, A., Ding, C., Li, D., et al.: Trust: triangle counting reloaded on gpus. IEEE Trans. Parallel Distrib. Syst.(TPDS) **32**(11), 2646–2660 (2021)
42. Park, H., Kim, M.S.: 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), pp 2051–2059 (2018)
43. Ribeiro-Junior, S., Quirino, R.D., Ribeiro, L.A., Martins, W.S.: Fast parallel set similarity joins on many-core architectures. J. Inf. Data Manag. **8**(3), 255–255 (2017)
44. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: Edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP), pp 472–488 (2013)
45. Sabet, A.H.N., Zhao, Z., Gupta, R.: Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In: Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys), pp 1–16 (2020)
46. Sato, H., Mizote, R., Matsuoka, S., Ogawa, H.: 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), pp 398–403 (2016)
47. Shi, T., Zhai, J., Wang, H., Chen, Q., Zhai, M., Hao, Z., Yang, H., Chen, W.: Graphset: High performance graph mining through equivalent set transformations. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–14 (2023)
48. Singh, D.P., Joshi, I., Choudhary, J.: Survey of gpu based sorting algorithms. Int. J. Parallel Program. **46**(6), 1017–1034 (2018)
49. Sun, S., Luo, Q.: Subgraph matching with effective matching order and indexing. IEEE Trans. Knowl. Data Eng. (TKDE) **34**(1), 491–505 (2020)
50. Teixeira, C.H., Fonseca, A.J., Serafini, M., Siganos, G., Zaki, M.J., Aboulnaga, A.: Arabesque: a system for distributed graph mining. In: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP), pp 425–440 (2015)
51. Wang, K., Zuo, Z., Thorpe, J., Nguyen, T.Q., Xu, G.H.: 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)), pp 763–782 (2018)
52. Wang, L., Wang, Y., Owens, J.D.: Fast parallel subgraph matching on the gpu. In: Proceeding of the International Symposium on High Performance Distributed Computing (HPDC) (2016)
53. Wang, Z., Meng, Z., Li, X., Lin, X., Zheng, L., Tian, C., Zhong, S.: Smog: Accelerating subgraph matching on gpus. In: 2023 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–7 (2023). https://doi.org/10.1109/HPEC58863.2023.10363569
54. Wei, H., Yu, J.X., Lu, C., Lin, X.: Speedup graph processing by graph ordering. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp 1813–1828 (2016)
55. Wei, Y.W., Chen, W.M., Tsai, H.H.: Accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using gpus. IEEE Trans. Parallel Distrib. Syst.(TPDS) **32**(9), 2352–2366 (2021)
56. Wolf, M.M., Deveci, M., Berry, J.W., Hammond, S.D., Rajamanickam, S.: Fast linear algebra-based triangle counting with kokkoskernels. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–7. IEEE (2017)
57. Wu, L., Liu, H.: 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), pp 637–645 (2018)
58. Yaşar, A., Rajamanickam, S., Berry, J.W., Çatalyürek, Ü.V.: A block-based triangle counting algorithm on heterogeneous environments. IEEE Trans. Parallel Distrib. Syst. (TPDS) **33**(2), 444–458 (2021)
59. Zeng, L., Zou, L., Özsu, M.T., Hu, L., Zhang, F.: GSI: GPU-friendly subgraph isomorphism. In: Proceedings of IEEE 36th International Conference on Data Engineering (ICDE), pp 1249–1260 (2020)
60. Zeng, Z., Wang, J., Zhou, L.: Efficient mining of minimal distinguishing subgraph patterns from graph databases. In: Proceeding of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp 1062–1068 (2008)
61. Zhang, J., Lu, Y., Spampinato, D.G., Franchetti, F.: FESIA: A fast and simd-efficient set intersection approach on modern cpus. In: 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, pp 1465–1476. IEEE (2020)
62. Zhang, Y., Liao, X., Jin, H., He, B., Liu, H., Gu, L.: Digraph: An efficient path-based iterative directed graph processing system on multiple gpus. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 601–614 (2019)
63. Zhao, C., Zhang, Z., Xu, P., Zheng, T., Guo, J.: Kaleido: An efficient out-of-core graph mining system on a single machine. In: Proceeding of the 36th International Conference on Data Engineering (ICDE), pp 673–684 (2020)
64. Zheng, T., Nellans, D., Zulfiqar, A., Stephenson, M., Keckler, S.W.: Towards high performance paged memory for gpus. In: Proceeding of the International Symposium on High Performance Computer Architecture (HPCA), pp 345–357 (2016)