Fast GPU-based Subgraph Search Using Parallel Vertex Matching

Journal:	*Transactions on Parallel and Distributed Systems*
Manuscript ID	TPDS-2023-01-0012
Manuscript Type:	Regular
Keywords:	Subgraph Search, Parallel Vertex Matching, Edge Label Partition, GPU

Fast GPU-based Subgraph Search Using Parallel Vertex Matching

Gangzhao Lu, Meng Hao, Weizhe Zhang, Hui He, Xiao Sun, and Zheng Wang

Abstract—Many graph-based workloads require performing *subgraph matching* by finding all subgraphs of a data graph that are isomorphic to an input query graph. Doing so on a real-life data graph is often computation-intensive because of the large number of graph vertices to be examined. Existing schemes for subgraph matching all adopt a simple scheme by matching vertices of a query graph one by one. This strategy fails to capitalize on the structure parallelism of graphs and can incur extensive memory accesses on GPUs. This work presents GENEVA, a new GPU-based subgraph matching scheme that can match multiple query vertices simultaneously. Unlike prior work, GENEVA performs subgraph matching within a single GPU kernel, eliminating many memory access operations required to process the intermediate results. GENEVA also provides an enhanced storage format to reduce the memory footprint of graph data and improve processing efficiency. We evaluate our approach by applying it to eight real-life graph datasets on an NVIDIA 2080Ti GPU. Experimental results show that our approach improves GSI, the state-of-the-art graph matching framework, by 80% on average (up to 96%), while reducing the memory footprint by 83%.

Index Terms—Subgraph Search, Parallel Vertex Matching, Edge Label Partition, GPU

1 INTRODUCTION

SUBGRAPH matching is a fundamental task of graph analysis. It requires finding all subgraphs from a data graph G that are isomorphic to a query graph q. Subgraph matching requires finding all the isomorphic subgraphs (known as graph embeddings[1]) from the data graph G by ensuring both the vertex and the edge labels of the extract subgraph matches the vertex and edge labels of the query graph q. This technique has a wide range of applications, including social network analysis [1], [2], and chemical compound search [3].

Subgraph matching is an NP-complete problem [4], requiring significant computation time when processing large, real-life graphs. A wide range of approaches have been proposed to speed up subgraph search [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. These approaches adopt a typical 3-step process for subgraph matching. For the vertices in the query graph q, this process first builds candidate sets and auxiliary data from the data graph G. It then determines a matching order that determines how each query vertex should be matched from the candidate sets and auxiliary data before performing the graph matching by following the matching order.

Some of the most recent works attempt to leverage the GPU computation power for fast graph matching [14], [6], [7], [10], [12]. GSI is the current state-of-the-art GPU-based graph matching algorithm [10], delivering the best performance on some of the representative datasets. It introduces a dedicated graph storage format to reduce the memory footprint and improve the performance for graph vertex matching. While promising, existing approaches can only match one vertex from the query graph in one go. Such a strategy leads to extensive GPU memory accesses because they need to load and store the intermediate results for each vertex matching. These memory operations are expensive on GPUs with sizeable intermediate results, which should be avoided if possible.

Most subgraph matching approaches do not explore data parallelism within a parallel thread because each thread only matches one vertex from the query graph in each iteration. Such a strategy leads to extensive GPU memory accesses because the GPU worker needs to load and store the intermediate results for each matched vertex. As we will show later in the paper, these memory operations are expensive on GPUs with sizeable intermediate results, leaving much room for performance improvement. While there are techniques for reducing the intermediate results on multi-core CPUs by simultaneously processing multiple graph edges [15], their strategy can handle a small set of vertice matching patterns. Our work aims to close this gap by enabling the simultaneous process of multiple vertices in one go to reduce the GPU memory accesses.

We propose GENEVA[2], a new GPU-based subgraph matching framework for parallel vertex matching within a GPU worker. GENEVA aims to match multiple vertices at each iteration and produce the corresponding results in one GPU kernel. Unlike prior work [15] that supports only matching pattern 0 in Fig. 1, our approach supports all seven representative matching patterns in Fig. 1. It uses one single GPU kernel to match multiple edges in a single GPU kernel

- *G. Lu, M. Hao, W. Zhang and H. He are with the School of Cyberspace Science at Harbin Institute of Technology, Harbin 150000, China. E-mail:{lugangzhao, haomeng, wzzhang, hehui}@hit.edu.cn, xiaosun@stu.hit.edu.cn*
- *Z. Wang is with the School of Computing at University of Leeds, United Kingdom. E-mail: z.wang5@leeds.ac.uk*

1. Not to confuse with the term embeddings (e.g., a vector of numerical values) used by deep neural networks.

2. GENEVA = subGraph sEarch usiNg parallEl Vertex mAtching.

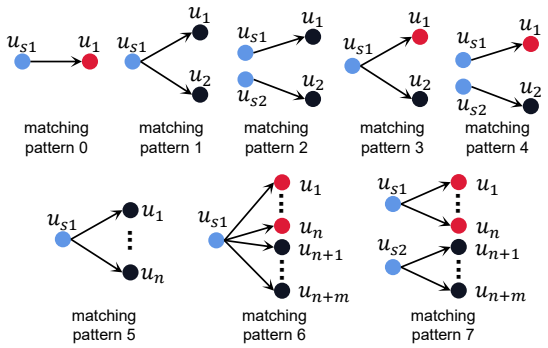

Fig. 1. Matching patterns supported by GENEVA. u_{s1} and u_{s2} are the query vertices to be matched, which can be of any vertex labels. $\{u_1, \cdots, u_n\}$ and $\{u_{n+1}, \cdots, u_{n+m}\}$ are the vertices from the data graph to be matched, and vertices in the same set have the same vertex label. Furthermore, edges in the the same matching pattern have the same edge label.

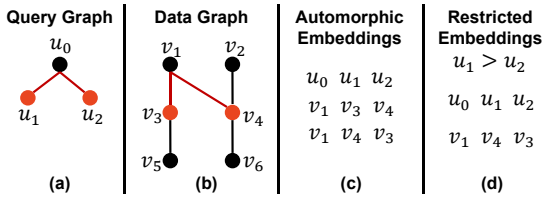

Fig. 2. Examples of automorphic and non-automorphic embeddings.

simultaneously. By matching multiple vertices and edges in a single GPU kernel, our approach eliminates the expensive GPU global memory addresses. GENEVA provides new optimizing algorithms to generate the vertex matching order and process the commonly used matching pattern. GENEVA also introduces a new graph storage format, which gives significant benefits for graph data storage overhead and processing time over GSI.

We evaluate GENEVA by applying it to eight real-world data graphs on an NVIDIA 2080Ti GPU. We compare GENEVA against GSI, the state-of-the-art GPU-based subgraph matching framework. Experimental results show that GENEVA reduces the processing time by $5\times$ on average over GSI. It uses 83% less memory for graph data storage and improves the vertex search time by 58%.

This paper makes the following technical contributions:

- It presents a novel parallel vertex matching method to support the process of multiple query vertices at the same time to reduce the GPU global memory access operations (Section 3);
- It presents an enhanced storage format to improve the storage efficiency and reduce the vertex search time (Section 4);
- It introduces an enhanced matching order generation algorithm to produce an appropriate vertex matching order to support efficient subgraph matching (Section 6).

2 BACKGROUND

2.1 Preliminaries

Given two graphs q and G, the task of subgraph matching is to determine if the larger graph G contains a subgraph that is isomorphic to the query graph q. If the query and data graphs include vertex and edge labels, both the topology and the labels should be matched. The large graph G is known as a data or target graph. Subgraph search is to find all subgraphs of the data graph, which are isomorphic to the query graph.

In this work, we target mining subgraphs from an *undirected* and *labeled* data graph $G = \{V, E, \Sigma, L_V, L_E\}$, where V, E and Σ are a set of vertices, edges and labels respectively, L_V is a function that associates a vertex $v \in V$ with a label $L_V \in \Sigma$, and L_E is a function that associates an edge $e \in E$ with a label $L_E \in \Sigma$. We describe a few important concepts of subgraph search as follows.

Embedding. Given a query graph q and a matching order π, GENEVA iteratively chooses one or multiple unprocessed vertices from the matching order π to perform subgraph search. Before we match the last vertex in the matching order, we only apply a sub-query graph. The subgraphs of G that are isomorphic to the query graph or sub-query graph are called subgraph isomorphic embeddings. A full subgraph isomorphic embedding is obtained if every vertex in the query graph q is mapped to a vertex in the data graph G. For example, for the query graph q and the data graph G in Fig. 3, there is one subgraph isomorphic embedding of q in G, which maps $(u_0, u_1, u_2, u_3, u_4, u_5, u_6)$ to $(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$ respectively. For simplicity, we use the term "*embedding*" to refer to "subgraph isomorphic embedding" thereafter.

Query graph structure. We follow the methdology in [9] to define the core structure of a query graph q. Here, the core structure is the minimal connected subgraph of q that contains all non-tree edges of q regarding any spanning tree of q. This structure is generated by iteratively removing all degree-one vertices from q. GENEVA uses the core structure to remove invalid embeddings, e.g. subgraphs whose topology or vertex or edge labels do not match q - see also Section 6.

Matching order. Given a query graph q, a matching order π is a permutation of vertices in q, where $\pi[i]$ is the ith vertex in π. In essence, the matching order defines which order we match individual vertices of the query graph q with the counterparts from the data graph G. Studies have shown that choosing the right matching order can have a significant impact on performance [9], [11], [13], [6]. GENEVA provides a dedicated algorithm to generate the matching order; see Section 6.

Matching patterns. There are different ways to match the query graph vertices with vertices from the data graph. Fig. 1 lists the matching patterns supported by GENEVA. For each pattern, GENEVA implements a pattern-specific matching algorithm. While different matching patterns can lead to the same subgraph matching outcomes, the processing overhead can vary depending on the pattern uses. Our strategy for choosing a matching pattern is described in Section 5.5.

Automorphism. Given a graph g, an automorphism of g is a match from g to itself, which also indicates that g is symmetric. Fig. 2 provides one of such examples, where the query graph has two symmetric vertices, u_1 and u_2, and

two isomorphic embeddings (Fig. 2c) that all corresponding to the same subgraph of the data graph with vertices v_1, v_3, v_4 (Fig. 2b). To eliminate the duplicate embeddings, the standard practice is to impose some restrictions on graph matching [16], [8]. Fig. 2d shows the matching criterion used by GraphPi and GraphZero, which chooses an embedding where the data graph candidate vertex that has the largest number (i.e., we choose to match u_1 with vertex v_4 rather than v_3 from the data graph) is chosen for a query graph vertex. This is a scheme used by GraphPi [8] and GraphZero [16], and GENEVA also adopts this common practice.

2.2 GPU Architecture

GPUs are massively parallel computing devices. They are widely used to accelerate graph processing tasks, including subgraph matching [10], [12], [6]. GPU processing units can be abstracted into a two-level hierarchy, the Streaming Multiprocessors (SMs) and computing cores inside the SM. An SM is further divided into processing blocks. Each processing block contains a fixed number of threads, called a warp that is the basic scheduling unit.

Modern GPUs also organize their memory into a hierarchical system, containing the global memory, a configurable shared memory, registers, and potentially an L2 cache between the global memory and the shared memory. The thread-local registers are the fastest memory component, having the lowest access latency (1-2 cycles). The SM local L1 caches and shared memory provide a larger storage capacity over the thread-local registers but have modestly higher accessing latency of around 30 cycles. Like the RAM in a CPU system, the GPU's off-chip global memory provides the largest memory storage capacity on the GPU but has the most expensive accessing latency of around 500 cycles.

The NVIDIA CUDA programming model provides atomic functions to perform a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. In this work, we use the CUDA $atomicAdd$ function. This function reads a variable in global memory before adding a number to it and then writes the result back to the same address.

3 Overview of Our Approach

Fig. 3 depicts the overall workflow of GENEVA. At the offline stage, we convert the data graph into a carefully designed storage format. This format is designed to accelerate GPU kernel computation while reducing the GPU memory footprint when processing large graphs (Section 4). We note that this storage conversion only needs to be performed once, which is a one-off cost performed offline.

During runtime, Algorithm 1 is used to perform subgraph search. We first decompose the query graph into the core structure and trees according to the method proposed in [9], and further decompose them into extension and elimination phases. Each extension phase contains exactly one matching pattern shown in Fig. 1 and each elimination phase contains one or more non-tree edges. In the meanwhile, we generate a matching order for extension and elimination phases, and restrictions on query vertices

Algorithm 1: SUBGRAPHSEARCH

Input: the data graph in our format G, the query graph q
Output: Embeddings of q EMB

```
1  π ← GenMatchOrder(q);
2  Load the edge label partition elp whose edge label is π[0].edgeLabel from G to GPU;
3  Allocate all available GPU memory space to newEMB;
4  ExtKernel(elp, NULL, newEMB, π[0]);
5  EMB ← newEMB;
6  for i ← 1 to π.size do
7      Load the edge label partition elp whose edge label is π[i].edgeLabel from G to GPU;
8      Allocate all available GPU memory space to newEMB;
9      if π[i] is an extension phase then
10         ExtGPUKernel(elp, EMB, newEMB, π[i]);
11     else
12         EliGPUKernel(elp, EMB, newEMB, π[i]);
13     EMB ← newEMB;
```

(line 1). All edges in an extension or elimination phase have the same edge label since we load one edge label partition at each iteration (lines 2, 7). The first extension phase of the matching order is used to generate initial embeddings (line 4). The main difference between the first and following extension phases is that the source vertices of the former one are from the edge label partition, while the later one are from previous embeddings. Finally, we use different GPU kernels to handle extension and elimination phases (lines 10, 12), and output final embeddings (line 13). In the following sections, we elaborate each step in detail.

4 Data Graph Storage Format

Real-world graphs are often too large to fit into the memory of a single GPU. Therefore, only part of the data that is needed for the current computation should be stored in the GPU memory at a given time. As a result, it is important to find a compact representation of the graph data without compromising the computation performance. To this end, GENEVA extends the Partitioned Compressed Sparse Row (PCSR), the data graph storage format used by GSI [10]. This format groups edges with the same label into an edge label partition, which is then stored in the Compressed Sparse Row (CSR) sparse matrix storage format. By doing so, only the partition with the same edge label as the matching edge needs to be moved from the CPU memory into the GPU memory. PCSR makes some small modifications to the CSR format. As depicted in Fig. 4a, a vertex ID (VID) can be used to find the row offset of a vertex because VIDs are stored as contiguous elements in a classical CSR. After grouping edges into partitions, VIDs of vertices are no longer guaranteed to be continuous in the storage space, GSI employs a hash function to translate a VID into a slot the corresponding elements in the edge partition structure, which we call hash-PCSR. To reduce collisions of the hash function, some empty entries have to be reserved for each vertex (30 in GSI), which results in a large portion of unused GPU memory space. GENEVA is designed to avoid this pitfall.

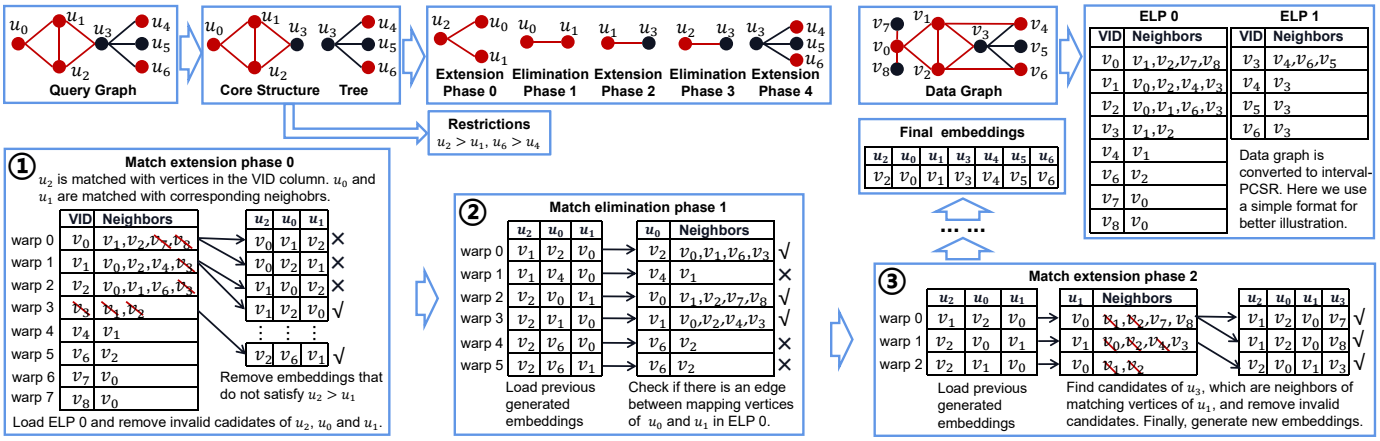

Fig. 3. An overview of our parallel vertex matching approach, GENEVA.

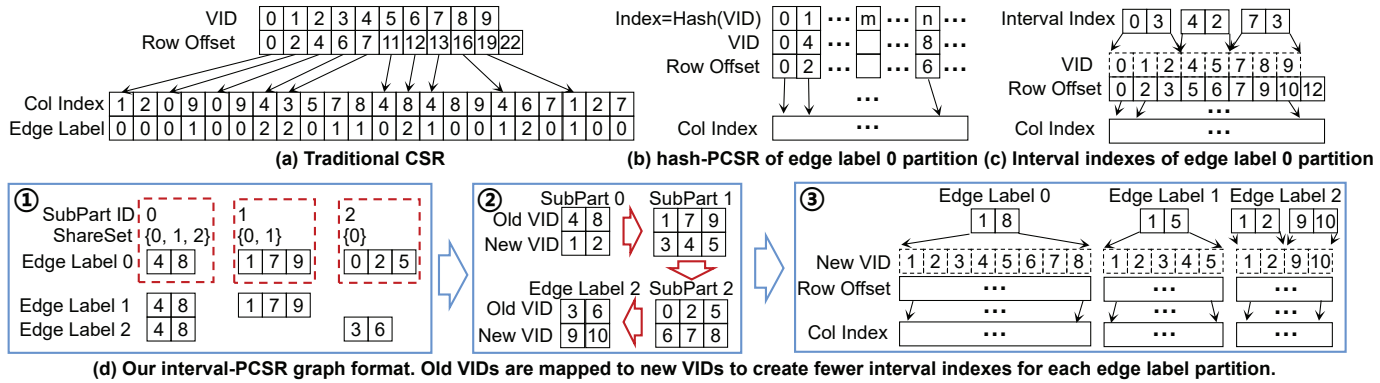

Fig. 4. Demonstration of data graph formats generated by traditional CSR (a), hash-PCSR that is used by GSI (b), non-optimized interval-PCSR (c), and our optimized interval-PCSR (d). Note that VIDs in dashed squares do not need to be stored in memory.

4.1 GENEVA Data Graph Storage Format

To reduce the amount of unused memory space required by the PCSR's hash function, we record the range of contiguous VIDs of an edge label partition (ELP). The contiguous range of VIDs is recorded in an *interval index* structure as shown in Fig. 4c, which stores the first VID and the number of continuous VIDs in the range.

Because an ELP may contain many small intervals, directly mapping each interval to be stored in an interval index will result in many interval indices. This is not ideal because having a large number of interval indices means we will need many memory accesses to just find the interval for a VID. For example, if we want to find the index of VID 7 in Fig. 4c, with a naïve interval scheme, we will have to compare number 7 against the first three interval indices before locating VID 7 in the third interval index. Our design avoids this pitfall by carefully mapping the original VIDs to new VIDs, to allow one to generate more contiguous new VIDs in each edge label partition, which in turn leads to a smaller number of intervals. We call the storage format of GENEVA interval-PCSR. Our data storage format shares the same spirit of Huffman coding [17] – we want to reduce the number of memory access when processing the largest ELP (i.e., the partition that contains the largest number of VIDs).

4.2 GENEVA Storage Format Algorithm

As described in Algorithm 2 and Fig. 4d, GENEVA takes several steps to convert the data graph into a GPU-tuned storage format, described as follows.

Step 1: Find the largest ELP. We start by choosing the partition that contains the most vertices (line 3). Our intuition is that the more vertices a partition contains, the higher probability it will be accessed. Therefore, reducing the number of intervals for this partition can help reduce the average memory access latency. For the example shown in Fig. 4d ①, ELP 0 is selected because it contains the largest number of vertices.

Step 2: Rename VIDs. In the second step, we try to increase the continuous VID interval by renaming the VIDs. To this end, for each vertex, we find out all ELPs that contain the vertex. For example, vertices 4 and 8 in Fig. 4d ① belong to ELPs 0, 1 and 2 - these ELPs form a share set, $shareSet$, for the two vertices. Next, we merge VIDs that have the same share sets into a subpartition, $subPart$ (line 4). For the example shown in Fig. 4d ①, we map vertices 4 and 8 to a subpartition, $subpart0$, because they have the same share set. We apply this grouping strategy to the largest ELP found in step 1. For the vertices that belong to the same subpartition, we can assign unique, continuous new VIDs to them in arbitrary order, as long as these new VIDs are contiguous and follow the largest VID from the last

Algorithm 2: GENMAP

```
Input: the set of all edge label partitions ELP
Output: the mapping array MAP with old and new
        VIDs are indices and values respectively
1  newVID ← 1;
2  while ELP ≠ ∅ do
3      Choose the partition elp ∈ ELP that has the most
       vertices;
4      Divide elp into sub-partitions subParts;
5      foreach subPart ∈ subParts do
6          Group IDs of partitions that contain subPart
           into shareSet and delete vertices of subPart
           from these partitions;
7      while subParts ≠ ∅ do
8          Choose the subPart ∈ subParts that is shared
           by the most partitions and delete it from
           subParts;
9          Construct MAP (assign new vertex IDs starting
           from newVID to vertices in subPart
           contiguously);
10         newVID ← newVID + subPart.size;
11         preSubPart ← subPart;
12         while preSubPart ≠ ∅ do
13             Choose the subPart ∈ subParts whose
               shareSet has the most same partition IDs
               with the shareSet of preSubPart and
               delete the subPart from subParts;
14             if Found the subPart then
15                 Construct MAP;
16                 newVID ← newVID + subPart.size;
17                 preSubPart ← subPart;
18             else
19                 preSubPart ← ∅;
20                 break;
21     ELP ← ELP/elp;
```

processed subpartition.

Step 3: Process subpartitions. To determine the order of the subpartitions of an ELP to be processed, we start from the subpartition that is shared by most ELPs (line 8). This means for the example shown in Fig. 4d ①, we would first process $subpart0$, because it has the largest share set (with 3 ELPs). The next subpartition we choose will be the one that has the largest number of common ELPs between its share set and the share set of the last chosen subpartition (line 13). Looking at Fig. 4d ① again, $subPart1$ will be the secondly chosen subpartition because its share set has two ELPs $(0, 1)$ with the share set of $subPart0$ (the firstly chosen subpartition). We choose this strategy because of the following two reasons. First, the larger number of elements in common in the share sets of two subpartitions, the more ELPs will have the two subpartitions. Secondly, because we ensure the VIDs between two consecutively processed subpartitions are continuous, we can then merge the VIDs of the two subpartitions to form a contiguous interval to be stored in a single interval index. Using this strategy, we can use a single interval index to record the new VIDs of $subPart$ 0 and 1 in Fig. 4d ②. For each selected subpartition, we assign new contiguous VIDs to old VIDs in the selected subpartition (lines 9-10, 15-17).

Step 4: Repeat before Stop. We delete the processed vertices from an ELP, and repeat steps 1 to 4 for each ELP in turn. This process stops until all VIDs have been renamed and recored in the interval indices. This is illustrated in Fig. 4d ③.

Table 2 shows the storage size of GENEVA against the PCSR scheme used by GSI for eight data graphs. The GENEVA storage format reduces the storage size (and the GPU memory footprint) by 83%.

5 EXTENSION AND ELIMINATION

Like mainstream graph matching frameworks [7], [10], GENEVA applies a two-step approach to perform graph search. In the extension phase, GENEVA matches graph vertices and edge labels between the query and the data graph to generate embeddings (i.e., matched subgraphs). In the elimination phase, we remove embeddings that do not match the specified non-tree query edges.

5.1 The Extension Phase

For a given query graph and a data graph in the GENEVA storage format, we iteratively perform subgraph matching in multiple extension phases. Each extension phase processes one of the matching patterns depicted in Fig. 1. Previous works [10], [11] use the traditional single vertex matching (SV-match) scheme. This scheme requires extensive load and store operations to the GPU memory because it needs to write the intermediate results after matching a query vertex and read the same intermediate results before matching the next query vertex. GENEVA is designed to avoid this pitfall.

A key hurdle for performing subgraph search on GPUs is that the number of embeddings generated by a GPU warp (the basic GPU scheduling unit – see Section 2.2) is different, and we have to carefully compute the memory location used to store the newly generated embeddings for each warp to avoid write conflicts. Prior works address this problem by either generating the embeddings twice (in order to use the first generation to compute the store locations) [7] or need to access all embeddings twice to compute the store location [10]. GENEVA takes a different approach by utilizing the CUDA `atomicAdd` primitive inside the extension kernel to directly compute the write addresses across GPU wraps, eliminating the need of accessing or generating an embedding twice. This atomic instruction ensures only one wrap can update the same variable at any given time and hence avoids the race condition of determining the store location of the embeddings across wraps. While there is an overhead associated with `atomicAdd`, we found that this is not a severe problem in subgraph search because of the intrinsic irregularity of graph structures.

Algorithm 3 describes the overall workflow of the GENEVA extension phase. For each embedding (line 3), we first obtain the source VIDs from the embedding according to the matching pattern to be processed (line 4). Next, we search the source VIDs in the GENEVA interval-PCSR of the loaded ELP and extract their neighbors (line 5). Then, we remove invalid neighbors that either have wrong vertex labels or do not satisfy the restrictions (lines 6-7). Finally, we use different algorithms to generate new embeddings for different matching patterns (lines 8-15).

In the remainder of this section, we first describe embedding generation methods for the fundamental matching

Algorithm 3: EXTPHASEKERNEL

Input: the edge label partition elp, partial embeddings EMB, the extension phase $extP$, the starting address of new embeddings $newEMB$

```
1  totNum ← 0;
2  Load interval indexes of elp into shared memory;
3  foreach emb ∈ EMB do
4      Get source VIDs u_s1 and u_s2 from emb;
5      Search u_s1 and u_s2 in interval indexes and extract
         their neighbors ne1 and ne2, respectively.;
6      Remove neighbors that do not have the same vertex
         labels as u_1 and u_2 from ne1 and ne2, respectively;
7      Remove neighbors that do not satisfy the
         restrictions of u_1 and u_2 from ne1 and ne2,
         respectively;
8      if extP is matching pattern 0 then
9          We use the traditional SV-match method to
             generate embeddings for extP;
10     else if extP is matching pattern 1 then
11         OPTDOUEXT(emb, ne1, totNum, newEMB);
12     else if extP is one of matching patterns 2, 3, and 4
         then
13         DOUEXT(emb, ne1, ne2, totNum, newEMB);
14     else if extP is one of matching patterns 5, 6, and 7
         then
15         NEXT(emb, ne1, ne2, totNum, newEMB, extP);
```

Algorithm 4: DOUEXT

Input: the partial embedding emb, candidates of u_1 C_1, candidates of u_2 C_2, the number of newly written embeddings $totNum$, the starting address of new embeddings $newEMB$

```
1  Find the index i of the first valid vertex in C_1;
2  writePos ← atomicAdd(totNum, 1 × C_2.size);
3  for j ← 0 to C_2.size do
4      if C_2[j] ∈ emb then
5          Add j to the set boundry;
6          u_1 ← 0; u_2 ← 0;
7      else
8          u_1 ← C_1[i]; u_2 ← C_2[j];
9      Write the new embedding (emb, u_1, u_2) to the
         address pointed by newEMB + writePos;
10     writePos ← writePos + emb.size + 2;
11 i ← i + 1;
12 while i < C_1.size do
13     Load 32 candidates from C_1 into tmp; i ← i + 32;
14     Remove candidates that exist in emb from tmp;
15     writePos ← atomicAdd(totNum, tmp.size ×
         (C_2.size − boundry.size));
16     foreach 0 ≤ k < tmp.size and 0 ≤ j < C_2.size and
         j ∉ boundry do
17         if This is matching pattern 2 and C_1[k] = C_2[j]
             then
18             u_1 ← 0; u_2 ← 0;
19         else
20             u_1 ← C_1[k]; u_2 ← C_2[j];
21         Write the new embedding (emb, u_1, u_2) to the
             address pointed by newEMB + writePos;
22         writePos ← writePos + emb.size + 2;
```

patterns 2-4 of Fig. 1. We then describe our optimized embedding generation method for the matching pattern 1 of Fig. 1 before showing how our optimizations can be extended to matching patterns 5-7. The matching pattern 0 uses the tradition method to generate embeddings, thus we do not elaborate it in this work. In Section 5.5, we describe our approach for choosing which of the candidate matching patterns to use for subgraph matching.

5.2 Matching Patterns 2 to 4

For matching patterns 2-4, we employ a simple method described in Algorithm 4 to generate new embeddings. Our key idea is to iterate over all combinations of candidates of u_1 and u_2 in the data graph (denoted as C_1 and C_2 respectively) to construct new embeddings by appending each valid combination to a new copy of the current embedding emb. We consider a combination to be valid if none of the VIDs in the combination is presented in the current emb. To avoid checking this condition repeatedly, we first check this condition for all vertices in C_2 once (lines 3-4) and record indexes of vertices whose VIDs also exist in emb (line 5). To fully utilize C_2, we generate new embeddings while checking the condition. Once a vertex in C_2 is checked to be valid (lines 6-7), we assign this vertex and the first valid vertex in C_1 to u_2 and u_1, respectively (lines 1,8). Then, we store the new embedding (emb, u_1, u_2) to the corresponding address (lines 9-10).

At the beginning of Algorithm 4, we allocate space for new embeddings generated when checking conditions for C_2 (line 2). Since we do not know the number of valid vertices in C_2 ahead of time, we allocate space to store the maximum sized embedding – the number of vertices in C_2 (denoted as $C_2.size$). If there are invalid vertices in C_2, we assign 0 to u_1 and u_2 to indicate invalid embeddings (line 6). During processing, the GPU kernel then ignores embeddings that contains a zero.

In Algorithm 4, after finding invalid vertices in C_2, we generate new embeddings for the rest vertices in C_1 (lines 11-12). In each iteration, we load 32 vertices in C_1 into the shared memory buffer tmp and remove invalid ones from it (lines 13-14). To allocate space for new embeddings, we estimate its count as the number of combinations of tmp and valid vertices in C_2 (line 15). Then, we generate new embeddings for these combinations (line 16). As u_1 and u_2 have the same vertex label in extension pattern 2, it is possible that two candidates of u_1 and u_2 are identical (line 17), leading to an invalid new embedding. To solve the problem, we assign 0 to u_1 and u_2 if the embedding is invalid (line 18); otherwise, we assign the corresponding candidates to u_1 and u_2 (line 20). Finally, we store the new embedding to the specified address (lines 21-22).

5.3 Matching Pattern 1

Algorithm 4 is ill-suite for matching pattern 1. For this matching pattern, u_1 and u_2 are extended from the same source vertex u_{s1} and have the same vertex label, i.e., $C_1 = C_2$. If Algorithm 4 is applied to this pattern, each vertex in C_2 can be accessed by up to $C_1.size$ times. As shown in Fig. 5a, element 1 is accessed at each iteration of i. If $C_1.size > 32$, we need to reload element 1 from global memory each time we access it. Accessing other elements also have the same problem.

GENEVA is designed to accelerate the matching process of matching pattern 1. We achieve this by rearranging the

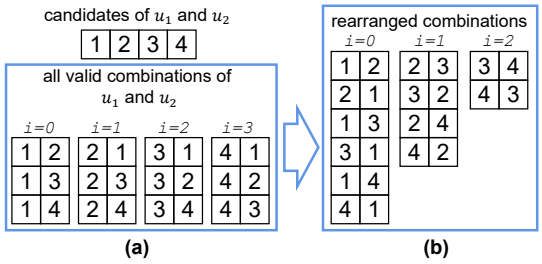

Fig. 5. An example of generating new embeddings for matching pattern 1.

Algorithm 5: OPTDOUEXT

Input: embeddings emb, candidates C, the number of newly written embeddings $totNum$, the starting address of new embeddings $newEMB$

1 $writePos \leftarrow atomicAdd(totNum, C.size \times (C.size-1))$;
2 **for** $i \leftarrow 0$ **to** $C.size - 1$ **do**
3 **for** $j \leftarrow i+1$ **to** $C.size$ **do**
4 Write new embeddings (emb, $C[i]$, $C[j]$) and (emb, $C[j]$, $C[i]$) to the address pointed by $newEMB + writePos$;
5 $writePos \leftarrow writePos + (emb.size + 2) \times 2$;

Algorithm 6: NEXT

Input: embeddings emb, candidates of $\{u_1, \cdots, u_n\}$ C_1, candidates of $\{u_{n+1}, \cdots, u_{n+m}\}$ C_2, the number of newly written embeddings $totNum$, the starting address of new embeddings $newEMB$, extension phase $extP$

1 **if** *$extP$ is matching pattern 5* **then**
2 **foreach** *$comb1$ of combinations of $u_1, \cdots, u_{n-2}$* **do**
3 $emb_{copy} \leftarrow (emb, comb1)$; $C_{copy} \leftarrow C_1/comb1$;
4 OPTDOUEXT($emb_{copy}, C_{copy}, totNum, newEMB$);
5 **else if** *$extP$ is one of matching patterns 6 and 7* **then**
6 **foreach** *$comb2$ of combinations of $u_{n+1}, \cdots, u_{n+m}$* **do**
7 **foreach** *$comb1$ of combinations of $u_1, \cdots, u_{n-2}$* **do**
8 $emb_{copy} \leftarrow (emb, comb1, comb2)$; $C_{copy} \leftarrow C_1/comb1$;
9 OPTDOUEXT($emb_{copy}, C_{copy}, totNum, newEMB$);

generation order of combinations, as shown in Fig. 5b. This arrangement promotes register usage, which in turn reduces the memory accessing latency. The core design of our optimized generation method for matching pattern 1 is shown in Algorithm 5. From Fig. 5b, we make two important observations to guide the design of Algorithm 5. First, since the element $C_1[i]$ is only used at iteration i, we will load it into a register (line 2) to reduce its access latency. Second, each time we generate a combination, we can reverse the combination to obtain another combination immediately without reloading data from global memory (lines 4-5).

5.4 Matching Patterns 5 to 7

Our approach for generating embeddings of matching patterns 5-7 is described in Algorithm 6. This algorithm extends Algorithm 5. For matching pattern 5 (line 1), we iterate over all combinations of $u_1, \cdots, u_{n-2}$ (line 2) and use Algorithm 5 to match the last two query vertices u_{n-1} and u_n (line 4). Before invoking Algorithm 5, we need to construct a new embedding and new candidates for u_{n-1} and u_n (line 3). For matching patterns 6 and 7, we first use an outer loop to iterate over all combinations of $u_{n+1}, \cdots, u_{n+m}$, and use the same method as the matching pattern 5 to generate embeddings (lines 7-9).

5.5 Priorities of Matching Patterns

When decomposing the query graph into extension and elimination phases, there is a possibility that multiple matching patterns are available for an extension phase. To choose the most suitable matching pattern, we apply a 2-level priority system to matching patterns. In the first level, we prioritize matching patterns based on the number of query vertices to be matched in each matching pattern. Thus, matching patterns 5-7 get the highest priority, 1-4 get the medium priority, and 0 gets the lowest priority. In the second level, we prioritize matching patterns that posses the same first level priority based on efficiency of each matching pattern when generating embeddings.

We first analyze the efficiency of medium priority matching patterns, and then the highest priority ones. The matching pattern 1 uses an optimized method (Algorithm 5) to generate embeddings, thus it is most efficient among matching patterns 1-4. Matching patterns 2-4 use the normal method (Algorithm 4) but the matching pattern 2 needs to evaluate the equality of candidates of u_1 and u_2. Therefore, the matching pattern 2 is the least efficient one. Compared to the matching pattern 3, the matching pattern 4 needs one more step to find the address of u_{s2}, thus it is less efficient than the matching pattern 3. Consequently, the priority order of matching patterns 1-4 from the highest to the lowest is matching pattern 1, 3, 4, and 2.

Among matching patterns of the highest priority, the matching pattern 5 is the most efficient one because it only needs one step to find the neighbor address of u_{s1} in the edge label partition and another step to find the address of neighbors that have the same vertex label as $u_1, \cdots, u_n$. In contrast, the matching pattern 7 is the least efficient one because it needs one more step to find the neighbor address of u_{s2} compared to the matching pattern 5. Consequently, the priority order of matching patterns 5-7 from the highest to the lowest is matching pattern 5, 6, and 7.

5.6 The Elimination Phase

In the elimination stage, GENEVA removes illegible embeddings that do not match the specified non-tree query edges. As shown in Figure 3 ②, the embedding (v_1, v_4, v_0) is removed because there is no edge between v_4 and v_0 that can match the query edge (u_0, u_1).

GSI [10] matches only one query edge in a GPU kernel and Lai et al. [15] matches at most two query edges at each iteration. Both methods can not fully utilize the elimination power of non-tree edges. In our approach, we match as many non-tree edges as possible to eliminate invalid embeddings at early stages. Algorithm 7 demonstrates how our approach deals with the eliminate phase. For each embedding emb (line 2), we first check if all query edges in the elimination phase can be matched in emb (line 4). If

Algorithm 7: ELIPHASEKERNEL

Input: the edge label partition elp, embeddings EMB, the number of generted embeddings $totNum$, the elimination phase $eliPhase$

1 Load interval indexes of elp into shared memory;
2 **foreach** $emb \in EMB$ **do**
3 **foreach** $edge \in eliPhase$ **do**
4 Check if there is an edge in emb that can match $edge$;
5 **if** *all edges in eliPhase are matched* **then**
6 Write emb to shared memory tmp;
7 **if** tmp *is full* **then**
8 $writePos \leftarrow atomicAdd(totNum, tmp.size)$;
9 Write tmp to the address pointed by $newEMB + writePos$;

so (line 5), we write emb to the shared memory tmp (line 6) and write tmp to global memory if it is full (lines 7-9).

6 MATCHING ORDER

We use the same method as [9] to decompose the query graph into the core structure and trees, as shown in Fig. 3. We also match the core structure first, and then the trees. The main difference is that our approach is designed specifically for parallel vertex matching.

When decomposing the core structure into extension and elimination phases, we need to adhere to an important constraint, which is that only matching patterns 0-4 can be used to decompose the core structure. The reason is explained as follows. In order to eliminate invalid embeddings as soon as possible, we need to first match circles in the core structure like $\{u_0,\ u_1,\ u_2\}$ in Fig. 3. Therefore, we only need to match at most two vertices when matching a circle because vertices in a circle have exactly two neighbors. Figure 3 demonstrates that the core structure is decomposed into matching patterns 0 and 1. We can see in the core structure that u_0, u_1, u_2, and u_3 can be matched with the matching pattern 6 where u_2 is the source vertex. If we first decompose the core structure with the matching pattern 6, and then the non-tree edge $(u_0,\ u_1)$, a large number of invalid embeddings may be generated, which significantly slows down the matching performance. After matching the core structure, we can use any appropriate matching patters to match trees. For example, the matching pattern 6 can be used to match the tree in Fig. 3.

Based on [9] and our parallel vertex matching method, we design Algorithm 8 to generate matching order of extension and elimination phases. At the beginning of Algorithm 8, we adopt methods proposed in [8], [16] to generate restrictions on VIDs in embeddings (line 1). Thus, we can avoid generating automorphic embeddings. Then we generate the core structure of q by iteratively removing degree-one vertices from q, and construct trees of q using removed vertices (line 2).

To match the core structure, we first find the smallest circle in the core structure (line 3), and then select vertices in the circle that conform to the highest priority matching pattern among matching patterns 0, 1, and 3 (line 4). The selected vertices constitute the first extension phase (lines 5-6), which is a special kind of extension phase. Different from the normal extension phase that fetches source vertices from embeddings (Fig. 3 ③), the first extension phase fetches the source vertex from the edge label partition (Fig. 3 ①). After generating the initial phase, we iteratively find vertices in the circle that conform to matching patterns 0-4 and choose the one with the highest priority (lines 7-9). Finally, we use the non-tree edge in the circle to construct an elimination phase (line 10).

After decomposing the smallest circle, we iteratively construct extension and elimination phases for the rest vertices and edges in the core structure (line 11). First, we find all non-tree edges and group them by the edge label (lines 12-13). For each group, we construct an elimination phase (lines 14-15). If no non-tree edges are found, we search for unmatched vertices in the core structure that can form one of matching patterns 0-4, and choose the one with the highest priority (lines 16-18).

After matching the core structure, we can use any appropriate matching patterns in Fig. 1 to match rest vertices that are not in the core structure since there is no non-tree edge left. At each iteration, we find vertices that can form matching patterns 0-7 and select the one with the highest priority (lines 19-21).

Algorithm 8: GENMATCHORDER

Input: the query graph q
Output: the match phase queue $matchPhase$

1 Generate restrictions to eliminate automorphisms;
2 Generate the core structure C and trees T of q;
3 Find the smallest circle $minc$ in C;
4 Select vertices in $minc$ that conform to matching patterns 0, 1, and 3, and choose the highest priority matching pattern;
5 Construct the initial phase with vertices corresponding to the selected matching pattern;
6 Add the initial phase to $matchPhase$;
7 **for** *vertices in* $minc$ *that conform to matching patterns 0-4* **do**
8 Choose the highest priority matching pattern and construct an extension phase;
9 Add the extension phase to $matchPhase$;
10 Construct an elimination phase with the non-tree edge in $minc$ and add it to $matchPhase$;
11 **for** *rest vertices and edges in* C **do**
12 **if** *there are non-tree edges* **then**
13 Group all non-tree edges by the edge label;
14 **foreach** *group* g **do**
15 Construct an elimination phase $eliPhase$ based on g and add it to $matchPhase$;
16 **else**
17 Find vertices that conform to matching patterns 0-4 and choose the one with the highest priority;
18 Construct an extension phase and add it to $matchPhase$;
19 **for** *vertices in* T **do**
20 Find vertices that can form matching patterns 0-7 and select the one with the highest priority;
21 Construct an extension phase and add it to $matchPhase$;

TABLE 1
Data graphs.

Type of sizes	Graph Name	$\|V\|$	$\|E\|$	$\|L_V\|$	$\|L_E\|$
Tiny	Enron	36K	183K	3	3
Tiny	FirstMM	56K	126K	3	3
Small	DD	0.3M	0.8M	5	5
Small	Gowalla	0.2M	0.9M	5	5
Medium	Patents	3.7M	16M	7	7
Medium	Reddit	4.6M	5.5M	7	7
Large	Orkut	3M	117M	12	12
Large	sinaweibo	58M	261M	12	12

7 Experimental Setup

7.1 Hardware and Workloads

Experimental platform. We evaluate our approach on a multi-core workstation with an NVIDIA RTX2080Ti GPU. The server has a 12-core Intel Xeon E5-2697 CPU at 2.3 GHz and 256 GB of RAM. The GPU has 11 GB of memory and 68 SMs where each SM has 4350 CUDA cores and 64KB of shared memory. Our evaluation system runs Ubuntu 16.04 with Linux kernel 4.15. We use gcc version 7.5 as the host compiler and NVIDIA CUDA toolkit version 11.0.

Data graphs. We use eight real-world data graphs in our experiment. The size of the graphs ranges from tiny to large, as illustrated in Table 1. Graphs Enron [18], Gowalla [19], Patents [20], and Orkut [21] are obtained from the SNAP dataset [22], while FirstMM, DD, Reddit, and sinaweibo are obtained from the Network Repository [23], [24].

Query graphs. Like GSI, we perform random walk on a data graph to extract query graphs for this data graph.

7.2 Evaluation Methodology

Competing methods. We compare GENEVA against GSI [10], the state-of-the-art GPU-based subgraph search method. We also provide an implementation variant of GENEVA, which matches one vertex at a time. This SV-match scheme is implemented by splitting each extension phase that contains one of matching patterns 1-7 in Fig. 1 into multiple extension phases that contain only the matching pattern 0.

Performance report. To measure the runtime of an approach, we run each test case at least five times and compute the 95% confidence interval bound. We increase the number of profiling runs if the interval is greater than 2%. We then report the geometric mean across runs.

8 Experimental Results

In this section, we first show that GENEVA improves the subgraph matching performance by $5\times$ over GSI (Section 8.1). We then show that our interval-PCSR reduces the storage overhead by 83% on average over the hash-PCSR format used by GSI while improving the processing time by $2.4\times$ for VID search (Section 8.2). We then compare our parallel matching approach against the single-vertice matching scheme in Section 8.3.

8.1 Overall Performance

In this section, we present the overall performance of GENEVA and GSI on eight data graphs. For each data graph, we first use random walk to extract nine query graphs from it with vertex count ranging from 4 to 12, and then employ both methods on the data graph to match the generated nine queries. The runtime results are shown in Fig. 6. GSI runs out of GPU memory in data graphs Orkut and sinaweibo.

Our approach outperforms GSI in almost all test cases and achieves an average speedup of $5\times$. The maximum speedup of our approach over GSI can be up to $22.5\times$. As can be seen, GSI outperforms our approach in two test cases. The reasons can be explained as follows: (1) When matching $Q9$ in the data graph Enron, GSI finds 620K embeddings while GENEVA finds over 19M embeddings. GSI misses a vast number of embeddings, which makes it more faster than GENEVA; (2) When matching $Q12$ in the data graph FirstMM, GSI exits after matching three vertices because no embeddings can be matched, while GENEVA matches all vertices and edges of $Q12$ and finds one embedding. Therefore, GENEVA is slower than GSI in this test case.

In Fig. 6, the average speedup of GENEVA over GSI is $7\times$ for small queries ($Q4$-$Q7$), while it is $3\times$ for large queries ($Q8$-$Q12$). The reason behind this phenomenon is explained as follows. GENEVA can generate an initial extension phase with at least two vertices. Additionally, if there exist vertices that form one of matching patterns 1-4, our approach can generate an initial extension phase with three vertices. Therefore, only two or three extension phases are needed for small queries to complete the matching process. This saves a large number of global memory accesses, which significantly accelerate the matching process for small query graphs.

To find out dominating factors for the superiority of GENEVA over GSI, we analyze detailed runtimes of each procedure of both methods. We employ GENEVA and GSI to match the query graph $Q4$ in the data graph Patents, and present the runtime of each step in Fig. 7. We only show runtimes of procedures invoked when matching the first two query vertices, u_0 and u_1. The reasons are twofold: (1) In GSI, steps after matching u_0 and u_1 repeat steps 6-9 of Fig. 7, and exhibit similar performance; (2) The runtimes of following steps are affected by the number of intermediate embeddings. Only when matching the first two vertices, we can make sure that both GENEVA and GSI process the same number embeddings.

We show the runtimes GENEVA and GSI in Fig. 7, and reveal three root causes of inefficiency of GSI compared to GENEVA. GSI utilizes a filter procedure to prune candidates for each query vertex. It takes twice of the runtime of the matching procedure of GENEVA. Moreover, GSI needs to transfer the signature table to GPU before running the filter procedure. The runtime of data transfer along takes up 68% runtime of GSI. This is because the signature table uses 64 bytes for each vertex in the data graph to store neighbor information, which results in a large data structure. Compared to GSI, GENEVA does not employ the filter procedure due to its inefficiencies in time and space.

Before the matching procedure, both methods need to transfer the edge label partition to GPU. However, the

Fig. 6. Execution times of GENEVA and GSI for searching nine query graphs on each of eight data graphs.

Geneva

Step 1. Transfer the label partition to GPU:	3.15 ms
Step 2. Invoke GPU kernel to generate embeddings:	0.87 ms
The matching procedure: 0.87 ms	

GSI

Step 1. Transfer the signature table to GPU:	47.36 ms
Step 2. Invoke GPU kernel to filter candidates for u_0:	0.54 ms
Step 3. Invoke GPU kernel to rearrange candidates:	0.50 ms
Step 4. Invoke GPU kernel to filter candidates for u_1:	0.68 ms
Step 5. Invoke GPU kernel to rearrange candidates:	0.24 ms
The filter procedure: 1.96 ms	
Step 6. Transfer the edge label partition to GPU:	18.20 ms
Step 7. Invoke GPU kernel to match u_1:	0.19 ms
Step 8. Invoke GPU kernel to rearrange candidattes:	0.11 ms
Step 9. Invoke GPU kernel to match the edge (u_0, u_1):	1.01 ms
The matching procedure: 1.31ms	

Fig. 7. The execution times of main procedures of GENEVA and GSI.

execution time of data transfer in GSI is 6x times slower than GENEVA. The reason is explained as follows. GSI builds a hash-PCSR data structure for each edge label partition and inserts 30 empty entries into hash indexes to reduce the number of collisions. Consequently, hash-PCSR occupies a large portion of memory space, and thus takes a much longer time to be moved to GPU than the interval-PCSR of GENEVA.

In the matching procedure, GSI invokes three GPU kernels to match the query vertex u_1, rearrange intermediate embeddings, and match the edge between u_0 and u_1, respectively. The kernel invoked in step 9 consumes most of the runtime of the matching procedure because it uses multiple time-consuming operations, such as thread block level synchronization and memory copy operations, inside the GPU kernel to maintain load balance. The four-layer load balance scheme of GSI can reduce the load imbalance greatly but also incur significant overhead. Different from GSI, our load balance scheme is simple yet effective. Only when there is no embeddings left, load imbalance can occur.

TABLE 2
Space overhead of CSR, hash-PCSR (GSI), and our interval-PCSR storage formats. The *Max* column represents the size of maximum edge label partition in the interval-PCSR. The *reduction over hash-PCSR* column represents the percentage of the space saved given by our interval-PCSR over GSI's hash-PCSR.

Graph	CSR	interval-PCSR	hash-PCSR	*Max*	*Reduction over hash-PCSR*
Enron	1.6MB	1.7MB	13MB	0.6MB	87%
FirstMM	1.2MB	1.5MB	22MB	0.5MB	93%
DD	15MB	11MB	139MB	2.1MB	87%
Gowalla	8.1MB	9.4MB	74MB	1.9MB	94%
Patents	149MB	184MB	1.6GB	26MB	89%
Reddit	60MB	71MB	901MB	10MB	92%
Orkut	906MB	997MB	1.94GB	100MB	50%
sinaweibo	2.2GB	2.5GB	10GB	256MB	75%

Overall, GENEVA significantly outperforms GSI.

8.2 Evaluation of Data Graph Formats

In this section, we first present the space cost of three data graph formats, including CSR, hash-PCSR, and interval-PCSR, and then the searching time of hash-PCSR and interval-PCSR.

The results of space cost are shown in Table 2. We can see from Table 2 that CSR has the smallest space among three data graph formats. The reason is that all VIDs in CSR are contiguous and unique, while VIDs in PCSR are non-contiguous and many duplicate VIDs exist in different edge label partitions. However, we need to transfer the whole CSR format graph to GPU in order to perform subgraph search, which is space and time consuming. In contrast, our interval-PCSR achieves similar space cost as CSR but only needs to transfer one edge label partition to GPU before each iteration of subgraph search. AS shown in Table 2, the size of the maximum edge label partition in interval-PCSR (denoted as *Max*) is greatly smaller than CSR, which makes data transfer in our approach more efficient.

Our interval-PCSR can averagely reduce the space cost of hash-PCSR by 83%. The main reason for the large space cost of hash-PCSR is empty entries in hash indexes. The hash-PCSR groups edges by their edge label, which makes VIDs in each group are non-contiguous. To find the index

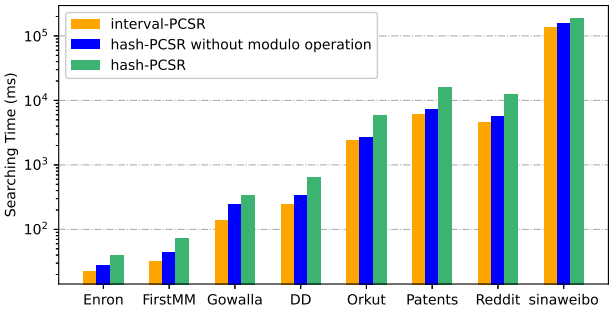

Fig. 8. The searching time of interval- and hash-PCSR on all data graphs.

of a given VID, hash-PCSR designs a hash function to map a given VID to a position that points to the neighbors of the VID. If multiple VIDs are mapped to the same position, a sequential search has to be performed to find the true position of the VID, which is a time-consuming operation in GPU. Therefore, hash-PCSR trades space for time and generates 30 empty entries for each VID to reduce collisions. This method significantly reduces collisions and also incurs large space cost. For example, the largest edge label partition in the data graph Enron contains 24737 VIDs. And hash-PCSR needs 32×24737 32-bit variables to store hash indexes. While in our interval-PCSR, we use one interval index to represent a range of contiguous VIDs and adopt Algorithm 2 to generate more contiguous VIDs. For the above example, we need only one interval index with two numbers to index positions of 24737 VIDs. The first number is the starting VID of this interval and the second number is the length of this interval. Thus, our interval-PCSR significantly reduces the space cost of hash-PCSR.

To further evaluate the effectiveness of our interval-PCSR, we compare the searching time spent on finding neighbors of a given VID between interval- and hash-PCSR. There is currently no applicable method that can measure the searching time inside a GPU thread without interfering the normal execution flow. Therefore, we extract searching related codes from GENEVA and GSI, and construct two GPU kernels using extracted codes respectively. We use the execution configuration of one thread block with one warp (32 threads) for both GPU kernels, which enables us to avoid interference such as warp scheduling. To ensure the accuracy of the measurement, we search all VIDs of a data graph in each edge label partition and record the searching time. The results are shown in Fig. 8.

We can see in Fig. 8 that our interval-PCSR achieves better performance than hash-PCSR in all data graphs and obtains an average speedup of 2.4×. In hash-PCSR, GSI first uses a hash function to calculate the index of a given VID, and then loads 32 indexes from global memory into shared memory. Finally, GSI finds the address of neighbors of the VID. Our approach needs to search the given VID in intervals that are stored in shared memory and load the address of neighbors from global memory. Though our approach needs more shared memory accesses than GSI, the accesses to shared memory in our interval-PCSR is aligned and the latency is negligible compared to the global memory access. The main reason for the overhead of hash-PCSR is the calculation of hash indexes. More specifically, the modulo operation in the hash function accounts for a large portion of execution time of the hash operation. We replace the variable divisor of the modulo operation with a fixed constant and present the searching time in Fig. 8. We can see that the hash-PCSR runs much faster without the modulo operation. However, variable divisor is necessary for the modulo operation to generate correct results.

In summary, our interval-PCSR not only reduces the space cost of hash-PCSR by replacing hash indexes with interval indexes, but also reduces the searching time by eliminating hash calculation.

8.3 Comparison of GENEVA and SV-match

To further evaluate the performance of GENEVA, we demonstrate how the number of extension phases that contain matching patterns 1-7 affect the performance of GENEVA. For simplicity, we call the extension phase that contains one of matching patterns 1-7 the PV-phase, and the extension phase that contains the matching pattern 0 the SV-phase. For each data graph, we use random walk to extract several query graphs and classify them into different categories by the number of PV-phase, and select one query graph from each category for each data graph. The results are shown in Fig. 9.

We can see in Fig. 9 that when the number of PV-phase is 0, both GENEVA and SV-match exhibit similar performance because all phases of both methods are the same. When the number of pV-phase is 1, 2, and 3, GENEVA improves the performance of SV-match by 11.3%, 20.2%, and 16.3% respectively. In the data graph DD, the performance of GENEVA is very close to SV-match because there is only one embedding that is isomorphic to the query graph and the runtime is too short to observe the difference between GENEVA and SV-match.

9 DISCUSSIONS

Naturally, there is room for further work and improvement. We discuss a few points there.

A more efficient embedding generation algorithm can be devised for matching patterns 5-7. In our present implementation, Algorithm 6, the efficient generation algorithm is only applied to iterate candidates of the last two query vertices, which ignores the fact that all query vertices share the same candidate set. Thus, we can use a similar method as Algorithm 5 to iterate over the candidate set of matching patterns 5-7.

A more efficient load balance scheme can be devised for subgraph searching. For now, we first generate the maximum number of warps that can run concurrently on a GPU, and then utilize the round robin method to distribute all embeddings to all warps. A drawback of this method is that some warps may finish their work early and have to wait for other warps since the time spent on processing each embedding is different. To overcome this problem, a work stealing method can be used when a warp finish its work early.

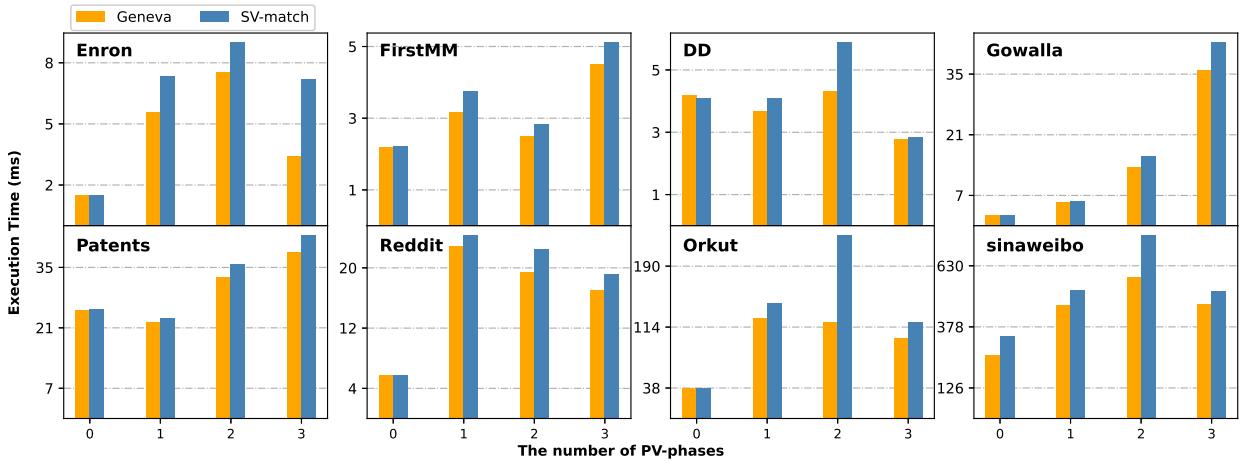

Fig. 9. Execution times of GENEVA and SV-match for different number of extension phases that contain matching patterns 1-7.

10 Related Work

10.1 CPU-based Subgraph Search

An early attempt for subgraph search is Ullmann [25], which uses a tree-search based approach to mine subgraphs. Many works [26], [27], [28], [29], [30] based on Ullmann have been proposed. VF2 [26] and QuickSI [27] utilize vertex and edge information to eliminate invalid embeddings. GADDI [28], GraphQL [29], and SPath [30] utilize the neighborhood information to remove unqualified data vertices from the candidate set for each query vertex. CECI [5] proposes a compact embedding cluster index to divide the data graph into embedding clusters and conduct subgraph matching on each embedding cluster.

Subsequent works [31], [32], [9], [33], [34] try to build auxiliary data to devise an effective matching order. TurboIso [31] first identifies candidate regions in a data graph and then generates a matching order for each candidate region based on the number of candidate vertices of query vertices in this candidate region. CFL [9] decomposes the query graph into the core-forest-leaf structure and matches the core structure first to eliminate invalid embeddings as early as possible. SGMatch [34] decomposes the query graph into graphlets and then generates a matching order based on graphlets. Distributed subgraph search has been explored in [35], [36], [8], [37], [38], [39], [40]. These approaches use MPI and MapReduce to distribute subgraph search tasks to different compute nodes.

Different from CPU-based approaches, our work focuses on exploiting a GPU's massive parallelism to match a query graph in a data graph.

10.2 GPU-based Subgraph Search

GPU has been used to accelerate applications in many fields because of its massive parallelism. There are several works exploiting GPU acceleration for subgraph search. TRICORE [41], Trust [42], TC-Stream [43] and [44] design a GPU-based triangle counting method. Guo at el. [6] partitions the data graph that beyond the GPU memory into subgraphs and matches the query graph in each subgraph iteratively. They also propose an embedding reuse method to avoid repeated computation in the work [12]. GSI [10], GSM [45], GpSM [7], and [14] utilize auxiliary data to assist candidate pruning and some GPU-based optimization techniques to speed up the matching process. While these works adopt the single vertex matching method, our approach uses parallel vertex matching method to reduce the number of read and write operations of embeddings.

11 Conclusions

We have presented GENEVA, a GPU-based parallel vertex matching scheme for performing subgraph search in a labeled undirected data graph. GENEVA is designed to match multiple vertices within a single GPU kernel to reduce the GPU memory accesses. It implements a new vertex storage format to reduce the memory footprint and vertex search time. We evaluate GENEVA by applying it to eight real-life graph datasets on an NVIDIDA 2080Ti GPU. We compare GENEVA against GSI, the state-of-the-art GPU-based subgraph search framework. Experimental results show that GENEVA consistently and significantly outperforms GSI on all test datasets. We also show that our parallel vertex matching scheme delivers better performance than a variant that implements a single-vertex matching scheme but with our enhanced storage format.

Acknowledgment

ACKNOWLEDGMENT This work was supported in part by the Joint Funds of the National Natural Science Foundation of China (Grant No. U22A2036) and in part by the National Natural Science Foundation of China (Grant No. 62202123). Prof. Zhang is the corresponding author.

References

[1] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, p. 812–823, May 2012.
[2] S. R. Kairam, D. J. Wang, and J. Leskovec, "The life and death of online groups: Predicting group growth and longevity," in *Acm International Conference on Web Search & Data Mining*, 2012.
[3] K. Wooyoung, L. Min, J. Wang, and P. Yi, "Biological network motif detection and evaluation," *BMC Systems Biology*, vol. 5, 2011.
[4] M. R. Garey and D. S. Johnson, "Computers and intractability: A guide to the theory of np-completeness," in *W. H. Freeman & Co.*, 1979.

[5] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1447–1462.

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

[7] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *International Conference on Database Systems for Advanced Applications*. Springer, 2015, pp. 299–315.

[8] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: high performance graph pattern matching through effective redundancy elimination," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[9] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1199–1214.

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

[11] S. Sun and Q. Luo, "Subgraph matching with effective matching order and indexing," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[12] W. Guo, Y. Li, and K.-L. Tan, "Exploiting reuse for gpu subgraph enumeration," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[13] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: a holistic approach to subgraph query processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 176–188, 2020.

[14] W. Lin, X. Xiao, X. Xie, and X.-L. Li, "Network motif discovery: A gpu approach," *IEEE transactions on knowledge and data engineering*, vol. 29, no. 3, pp. 513–528, 2016.

[15] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 974–985, 2015.

[16] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: Breaking symmetry for efficient graph mining," *arXiv preprint arXiv:1911.12877*, 2019.

[17] A. Moffat, "Huffman coding," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3342555

[18] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[19] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: user movement in location-based social networks," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1082–1090.

[20] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 177–187.

[21] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[22] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[23] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[24] ——, "An interactive data repository with visual analytics," *SIGKDD Explor.*, vol. 17, no. 2, pp. 37–41, 2016. [Online]. Available: http://networkrepository.com

[25] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[26] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, 2001, pp. 149–159.

[27] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[28] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009, pp. 192–203.

[29] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 405–418.

[30] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[31] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 337–348.

[32] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[33] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1429–1446.

[34] C. R. Rivero and H. M. Jamil, "Efficient and scalable labeled subgraph matching using sgmatch," *Knowledge and Information Systems*, vol. 51, no. 1, pp. 61–87, 2017.

[35] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 62–73.

[36] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 625–636.

[37] N. Talukder and M. J. Zaki, "A distributed approach for graph mining in massive networks," *Data Mining and Knowledge Discovery*, vol. 30, no. 5, pp. 1024–1052, 2016.

[38] S. Sun and Q. Luo, "Parallelizing recursive backtracking based subgraph matching on a single machine," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 1–9.

[39] T. Plantenga, "Inexact subgraph isomorphism in mapreduce," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 164–175, 2013.

[40] T. Reza, M. Ripeanu, N. Tripoul, G. Sanders, and R. Pearce, "Prunejuice: Pruning trillion-edge graphs to a precise pattern-matching solution," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 265–281.

[41] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 171–182.

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

[43] J. Huang, H. Wang, X. Fei, X. Wang, and W. Chen, "Tcstream: Large-scale graph triangle-counting on a single machine using gpus," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[44] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[45] L. Wang and J. D. Owens, "Fast gunrock subgraph matching (gsm) on gpus," *arXiv preprint arXiv:2003.01527*, 2020.

Gangzhao Lu received the B.S. and Ph.D. degree in computer science and engineering in 2014 and 2021 respectively from Harbin Institute of Technology, China. His research interests include performance modeling, parallel optimization, auto-tuning.

Meng Hao received the B.S. and Ph.D. degree in computer science and engineering from Harbin Institute of Technology, China, in 2014 and 2020 respectively. He is currently an assistant professor in the School of Cyberspace Science, Harbin Institute of Technology. His research interests include high-performance computing, performance modeling, and parallel optimization.

Weizhe Zhang (Senior Member, IEEE) received B.Eng, M.Eng and Ph.D. degree of Engineering in computer science and technology in 1999, 2001 and 2006 respectively from Harbin Institute of Technology.

He is currently a professor in the School of Computer Science and Technology at Harbin Institute of Technology, China, and director in the Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer network. He has published more than 100 academic papers in journals, books, and conference proceedings.

Hui He received the B.Eng, M.Eng, and Ph.D. degrees from the Harbin Institute of Technology, Harbin, China, in 1997, 1999, and 2006, respectively, all in computer science and technology. She is a Professor with the School of Cyberspace Science, Harbin Institute of Technology. Her current research interests include distributed computing, internet of things, and big data analysis.

Xiao Sun received the B.S. degree in computer science and engineering in 2021 from Harbin Institute of Technology, China. His research interests include performance modeling and parallel optimization.

Zheng Wang is an associate professor with the University of Leeds. His research focuses on parallel computing, compilation and systems security.