G-Morph: Induced Subgraph Isomorphism Search of Labeled Graphs on a GPU

Bryan Rowe and Rajiv Gupta(✉)

CSE Department, University of California, Riverside, Riverside, USA
{roweb,gupta}@cs.ucr.edu

Abstract. Induced subgraph isomorphism search finds the occurrences of embedded subgraphs within a single large data graph that are strictly isomorphic to a given query graph. Labeled graphs contain object types and are a primary input source to this core search problem, which applies to systems like graph databases for answering queries. In recent years, researchers have employed GPU parallel solutions to this problem to help accelerate runtimes by utilizing the filtering-and-joining framework, which first filters vertices that cannot be part of the solution then joins partial solutions with candidate edges until full isomorphisms are determined. However, the performance of current GPU-based solutions is hindered by limited filtering effectiveness and presence of extraneous computations. This paper presents G-Morph, a fast GPU-based induced subgraph isomorphism search system for labeled graphs. Our filtering-and-joining system parallelizes both phases by upfront eliminating irrelevant vertices via a novel space-efficient vertex signature hashing strategy and efficiently joining partial solutions through use of a novel sliding window algorithm (slide-join) that provides overflow protection for larger input. Together these techniques greatly reduce extraneous computations by reducing Cartesian products and improving edge verification while supporting large scan operations (split-scan). G-Morph outperforms the state-of-the-art GPU-based GSI and CPU-based VF3 systems on labeled real-world graphs achieving speedups of up to 15.78× and 43.56× respectively.

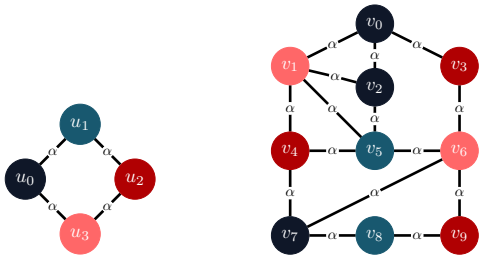
Keywords: Graph databases · Subgraph isomorphism · Breadth-first search · GPU programming · Parallel computing

1 Introduction

Labeled real-world graphs are ubiquitous and naturally arise as social networks [9], knowledge graphs [23], information networks [22], and various other types. These schemaless graphs are processed to extract insightful analytics in real time. Designing efficient, cost-effective graph analysis systems that process structurally diverse graphs has become an important research problem that is receiving considerable attention.

Given a query graph G_q and data graph G_d, one such graph analysis problem *subgraph isomorphism search* is deployed when one is interested in finding subgraphs of G_d that are isomorphic (i.e. same structure with labels) to G_q. Chemical substructure search [18], protein-protein interaction network search [5], and RDF graph databases [28] are select applications that benefit from subgraph search systems.

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 402–417, 2021.
https://doi.org/10.1007/978-3-030-85665-6_25



(a) Query graph G_q (b) Data graph G_d

m	u_0	u_1	u_2	u_3
1	v_2	v_5	v_4	v_1
2	v_7	v_8	v_9	v_6

(c) Table M matches

Fig. 1. Example with vertex labels (colors) & edge labels and results table M.

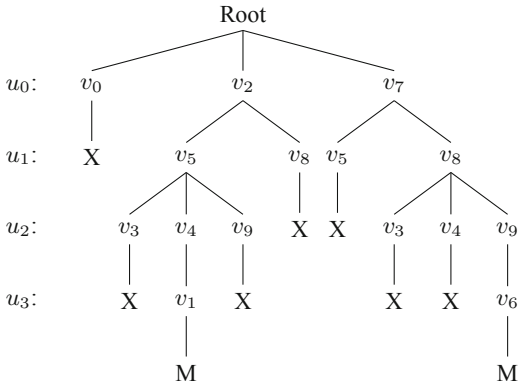


Fig. 2. Search tree for Fig. 1 graph with filtering based on vertex label per level. **X** is a pruned endpoint and **M** is a valid match.

Consider Fig. 1 with G_q, the query graph, and G_d, the data graph. The subgraph search system finds isomorphic subgraph matches M where rows are matches m with corresponding label-preserving vertex mappings $u \mapsto v$. This work handles both induced and non-induced versions of this search problem. In Fig. 1(c), the system would only return $m = 2$ if only *induced* matches are allowed since $m = 2$ does not contain extra edges between mapped data vertices. The match at $m = 1$ would be omitted because the non-edge (u_3, u_1) in G_q is not a valid match to the edge (v_1, v_5) in G_d.

Traditional subgraph isomorphism search algorithms [7,8,11,25,30] are generally based on a 2-phase *filtering-and-verifying* design [15]. The *filtering* phase performs vertex elimination to filter out initial nonsolution vertices. The *verifying* phase performs recursive backtracking to traverse a search tree (Fig. 2) such that each level of the tree represents a query vertex map and each path from the root to leaf in the tree represents a possible match. Existing algorithms explore novel methods of pruning the search tree.

Subgraph isomorphism search is NP-hard [21] and considered a computationally demanding problem to solve. In recent years, progress has been made to accelerate runtimes via deployment on distributed [20] and multithreaded platforms [6,17,19].

Table 1. Comparison of filtering-and-joining strategies.

System	Filter	Join (edges)	Join (M table)
GpSM [24]	Initialize-recurse	Verify-write	Verify-write
GSI [29]	Local indexing	N/A	Prealloc-combine
G-Morph	Local+cycle indexing	Slide-join	Verify-write

Specifically, the graphics processing unit (GPU) has shown to be an efficient and cost-effective platform to implement parallel graph algorithms [14,27]. With design principles of avoiding warp divergence [10] and favoring coalesced memory accesses, subgraph search on GPUs that run in single-instruction multiple data (SIMD) fashion has seen promising results with systems including GpSM [24], GunrockSM [26], and GSI [29].

GPU-accelerated systems are based on a 2-phase *filtering-and-joining* strategy where the *joining* phase is a parallelism friendly form of *verifying*. The *filtering* phase analyzes each vertex in G_d and determines via filtering strategies if data vertex v_i is a member of the query candidate set $C(u_j)$, that is, if $v_i \in C(u_j)$, and adds only relevant data vertex candidates to each query candidate set. The *joining* phase takes as input the candidate vertices and performs a parallel set join operation between adjacent candidate vertices to incrementally build valid partial matches until final matches M are realized.

Problems. Existing GPU-based systems propose different parallel strategies that handle both filtering and joining phases, but are prone to contribute to *extraneous computations* in both phases. Effective filtering of irrelevant vertices as well as efficient filtering in terms of time and space are key to eliminate extraneous search paths. The joining phase is prone to extraneous edge verification depending on algorithms used.

Current filtering strategies employed by GpSM and GSI (Table 1) present a tradeoff between filtering time and space required. GpSM's *initalize-recurse* suffers from multiple execution rounds but benefits from no metadata overhead. GSI's *local indexing* uses precomputed signatures to perform faster filtering but signatures used are prone to higher storage costs and only index immediate neighbors.

Joining consists of candidate edge and M table construction. The duplication of *edge verification*, the determination of adjacency between candidate vertices, when constructing candidate edges is a performance issue. GpSM's *verify-write* (Table 1) is employed in both joining parts, which involves duplicate verification by counting the join size before writing results to an allocated array. GSI's vertex-centric *prealloc-combine* avoids duplicate joins via preallocation, but lacks candidate edges for fast lookups.

Our Approach. In this paper we present **G-Morph**, an efficient GPU-based, filtering-and-joining subgraph isomorphism search system of labeled graphs, the first to support both *induced* and *non-induced* matches. Our design goal is to exploit labels to quickly eliminate irrelevant vertices and to improve edge verification, both of which limit extraneous computations. We propose *joining* phase algorithms that complement our *filtering* strategy while utilizing GPU parallelism and coalesced memory accesses.

We propose a novel space-efficient vertex signature *local+cycle indexing* strategy (Table 1) that leverages vertex properties of labeled graphs and stores them as hash

codes used for fast membership decisions. Compared with GpSM, *local+cycle indexing* quickly filters in one round. Comprised *local* (LOC) and triangle *cycle* (TRI) hash codes facilitate fast filtering with integer-wise metadata (a compact alternative of GSI).

We propose a novel sliding window algorithm *slide-join* for efficient joining, which builds candidate edges without duplicate edge verification (an improvement of GpSM) and offers overflow protection by segmenting the Cartesian product space. Utilizing efficient candidate edges with *slide-join* can lead to speedups over the GSI vertex-centric strategy. Additionally, *split-scan* is employed to support large *scans* [2].

Experiments show that our system achieves significant speedups over current state-of-the-art CPU- and GPU- based systems on labeled graphs. On labeled real-world graphs, G-Morph handily outperforms the best available GPU-based GSI and CPU-based VF3 algorithms delivering speedups of up to 15.78× and 43.56× respectively.

2 Preliminaries: Subgraph Isomorphism Search

Our goal is to find all subgraph isomorphisms, i.e. matches, of a single large data graph G_d such that each instance is *subgraph isomorphic* to a given query graph G_q. This paper considers *undirected labeled* graphs supporting both vertex and edge labels.

Definition 1 (Labeled Graph). A labeled graph is defined as 4-tuple $G = (V, E, L, l)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, L is a vertex labeling function, and l is an edge labeling function.

Definition 2 (Graph Isomorphism). Given two labeled graphs $G_a = (V_a, E_a, L_a, l_a)$ and $G_b = (V_b, E_b, L_b, l_b)$, G_a is *graph isomorphic* to G_b iff a *bijective* function $f : G_a \mapsto G_b$ exists such that:

(i) $\forall\ u \in V_a, \quad L_a(u) = L_b(\ f(u)\ )$
(ii) $\forall\ (u,v) \in E_a,\ (f(u), f(v)) \in E_b \ \wedge\ l_a(\ (u,v)\ ) = l_b(\ (f(u), f(v))\ )$
(iii) $\forall\ (f(u), f(v)) \in E_b,\ (u,v) \in E_a \ \wedge\ l_b((f(u), f(v))) = l_a((u,v))$

Definition 3 (Subgraph Isomorphism). Given two labeled graphs $G_q = (V_q, E_q, L_q, l_q)$ and $G_d = (V_d, E_d, L_d, l_d)$, G_q is *subgraph isomorphic* to G_d if an *injective* function $f : G_q \mapsto G_d$ exists such that:

(i) $\forall\ v \in V_q,\ \ f(v) \in G_d\ \wedge\ L_q(v) = L_d(f(v))$
(ii) $\forall\ (u,v) \in E_q,\ \ (f(u), f(v)) \in E_d\ \wedge\ l_q(u,v) = l_d(f(u), f(v))$

Graph and subgraph isomorphism are decision problems. However, practical real-world applications benefit more from listing the *matches* of subgraphs found via subgraph isomorphism *search*, which is the problem studied here.

Definition 4 (Subgraph Isomorphism Search). Given two labeled graphs $G_q = (V_q, E_q, L_q, l_q)$ and $G_d = (V_d, E_d, L_d, l_d)$, subgraph isomorphism search finds all matches, $m \in M$, of G_d, where a *match* is a representative subgraph of G_d that is *subgraph isomorphic* to G_q.

Table 2. Notation.

Symbol	Definition
$C\_Set(V_q)$	Total candidate set of all data vertices $\forall u \in V_q$
$C(u)$	Candidate set of data vertices for query vertex u
$N(u)$	Neighbor set of $u \in V$
$deg(u)$	Degree of vertex u
M', M	Partial matches table, Final matches table
$m \in M$	Subgraph isomorphism match instance
$freq(L(u))$	Count of unique vertices labeled by L
$\mathcal{L}, \lvert\mathcal{L}\rvert$	Set of vertex labels, Count of distinct vertex labels
$\ell, \lvert\ell\rvert$	Set of edge labels, Count of distinct edge labels
$VS\_List(V)$	Total vertex signature list $\forall u \in V$
$VS(u)$	Vertex signature of $u \in V$
$\vee, \oplus$	Bitwise OR, Bitwise XOR
$<< (n, m)$	Rotational left shift n by m bits
$\mathcal{E}$	Hash code in definitions

Definition 5 (Induced Subgraph Isomorphism Search). Given labeled graphs $G_q = (V_q, E_q, L_q, l_q)$ and $G_d = (V_d, E_d, L_d, l_d)$, induced subgraph isomorphism search finds all matches, $m \in M$, of G_d, where a *match* is a representative subgraph of G_d that is *graph isomorphic* to G_q.

The stricter *induced* version of the search problem prohibits extra edges between mapped vertices $\forall m \in M$ that do not appear in the G_q, i.e. $\forall m \in M$, there is a valid match of both edges and non-edges between G_q and m. Table 2 contains key notations.

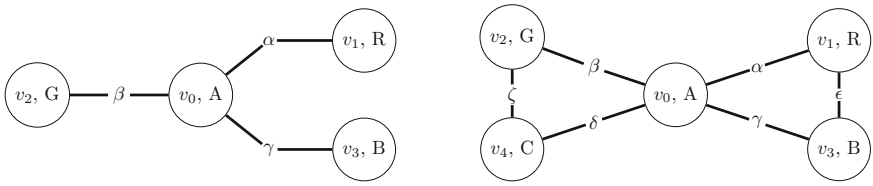
3 Vertex Signature Hashing

We define our space-efficient vertex signature $VS(u)$ used as our *local+cycle indexing* strategy, which comprises of two hash codes for high filtering power to help reduce extraneous computations in joining. We later discuss & evaluate our implementation.

3.1 Vertex Signature

The vertex signature encodes *local* and *cycle* properties of v, exploits surrounding $N(v)$ information for *local* filtering to extract data that is specific to $v \in V_d$, and is used for filtering purposes w.r.t. $u \in V_q$. Additionally, *cycle* properties are utilized to express refined filtering with triangles that *local* properties would not be able to exploit.

Definition 6 (Local Label Encoding - LOC). Given a vertex $u \in V$ and hash function f, a *local label encoding* (LOC) is an n-length bit-vector hash code $\mathcal{E}_{loc}$ that incorporates neighboring edge/vertex label pairs $\forall v \in N(u), l((u, v))$ and $L(v)$, s.t. $\mathcal{E}_{loc}$ is the bitwise $\vee$ result computed as:



(a) LOC example. (b) TRI example.

Fig. 3. Example depictions of LOC and TRI.

$$\mathcal{E}_{loc} = f(\, l(\, (u, v_0)\,), L(v_0)\,) \vee f(\, l(\, (u, v_1)\,), L(v_1)\,) \cdots \vee \\ f(\, l(\, (u, v_{|N(u)|-1})\,), L(v_{|N(u)|-1})\,).$$

Example 1 (Local Label Encoding). Given Fig. 3(a), where $L(v_0) = A$, $L(v_1) = R$, $L(v_2) = G$, $L(v_3) = B$, $l(\, (v_0, v_1)\,) = \alpha$, $l(\, (v_0, v_2)\,) = \beta$, and $l(\, (v_0, v_3)\,) = \gamma$, assume $n = 8$, vertex and edge labels map to integers, and hash function $f(x, y) =<< (1, y * |\ell| + x)$, the $\mathcal{E}_{loc}$ at vertex v_0 is:

$$\begin{aligned}\mathcal{E}_{loc} &= f(\, \alpha, R\,) \;\vee\; f(\, \beta, G\,) \;\vee\; f(\, \gamma, B\,) \\ &= [0,0,0,0,1,0,0,0] \;\vee\; [0,1,0,0,0,0,0,0] \;\vee\; [0,0,0,0,0,0,1,0] \\ &= [0,1,0,0,1,0,1,0]\end{aligned}$$

Definition 7 (Triangle Edge Encoding - TRI). Given a vertex $u \in V$ and hash function g, $\forall\, v \in N(v)$, a *triangle edge encoding* (TRI) is an n-length bit-vector hash code $\mathcal{E}_{tri}$ that encapsulates vertex and edge labels of immediate *triangles* of u such that u, a, and b form a *triangle*, which is a subgraph $\triangle$ where $a \in N(u)$, $b \in N(u)$, and there exists an edge, i.e. triangle edge, between a and b. $\forall \triangle \in N(u)$, $\mathcal{E}_{tri}$ is the bitwise $\vee$ result computed as:

$$\mathcal{E}_{tri} = g\,(\, L(a_0), L(b_0), l(\, (a_0, b_0)\,)\,) \vee g\,(\, L(a_1), L(b_1), l(\, (a_1, b_1)\,)\,) \cdots \vee \\ g\,(\, L(a_{|\triangle|-1}), L(b_{|\triangle|-1}), l(\, (a_{|\triangle|-1}, b_{|\triangle|-1})\,)\,).$$

Example 2 (Triangle Edge Encoding). Given Fig. 3(b), assume $n = 8$, vertex and edge labels map to integers, and hash function $g(x, y, z) =<< (1, (\, (x \oplus y) \bmod |\mathcal{L}|\,) * |\ell| + z)$, the $\mathcal{E}_{tri}$ at vertex v_0 is:

$$\begin{aligned}\mathcal{E}_{tri} &= g(G, C, \zeta) \;\vee\; g(R, B, \epsilon) \\ &= [1,0,0,0,0,0,0,0] \;\vee\; [0,0,0,0,0,0,0,1] \\ &= [1,0,0,0,0,0,0,1]\end{aligned}$$

Definition 8 (Vertex Signature). Given a vertex $u \in V$, a vertex signature is a 2-tuple $VS(u) = (\mathcal{E}_{loc}, \mathcal{E}_{tri})$ that represents a $|\mathcal{E}_{loc}| + |\mathcal{E}_{tri}|$-length bit-vector encoding of u.

3.2 Implementation and Evaluation

We store hash codes $\mathcal{E}_{loc}$ and $\mathcal{E}_{tri}$ as 64-bit unsigned integers regardless of $|\mathcal{L}|$ and $|\ell|$ as a general-purpose, space-efficient strategy using f and g previously defined. Data signatures $\forall v \in V_d$ are generated offline while query signatures $\forall u \in V_q$ are generated at runtime, which are compared against each other to determine $C(u)$ membership:

```
typedef unsigned long long ull64;
bool dvertex_is_candidate(ull64 online, ull64 offline) {
    return online & offline == online;
}
```

Our filtering in G-Morph (GM) differs from GSI [29] with size flexibility since GSI stores 2-bit counts per vertex sized $2|\mathcal{L}| * |\ell|$, which may become large if directly implemented with large unique label counts. GM guarantees matching vertex labels, which saves space, differs from [29,31], and is used to optimize joining. Our TRI filtering strategy is based on triangles rather than connecting edges [31] and utilizes the XOR operator, $|\mathcal{L}|$, and $|\ell|$ to evenly distribute possible input combinations.

We compared GM filtering against the filtering described in GSI. We used graphs identified in Table 4 (later in the paper) with 64-bit unsigned integers for both LOC and TRI, sized $|\mathcal{E}_{loc}|+|\mathcal{E}_{tri}| = 128$ bits, which equaled GSI's signature $2|\mathcal{L}|*|\ell| = 128$ bits with dataset $|\mathcal{L}| = 8$ & $|\ell| = 8$. Exact label and max degree filtering were applied with a uniquely labeled triangle query. Minimum candidate sizes (Table 3) were obtained.

Table 3. GM vs. GSI: comparing filtering effectiveness.

Graph	$\lvert V_d\rvert$	GM ($min(\lvert C(u)\rvert)$)	GSI ($min(\lvert C(u)\rvert)$)
CM	23k	37	59
AM	335k	132	392
DB	317k	442	695
EN	37k	139	167
BK	58k	98	164
GO	197k	570	745
FB	22k	158	181

GM filters more candidates in this scenario due to the distinct pair labeling of the query graph when signature sizes match. We see the addition of TRI is effective and helps filter additional candidates that would otherwise be unfiltered. We acknowledge that GSI may filter more vertices in other scenarios but with a space penalty of larger signature sizes.

4 G-Morph

In this section, we describe G-Morph, an efficient GPU-based induced subgraph isomorphism search system. We begin with a high-level overview then transition to

```
1  void SubgraphSearch(graph q_graph, graph d_graph,
2          siglist offline_vertex) {
3      /* Generate Signatures */
4      siglist online_vertex(q_graph);
5      /* Filtering */
6      set c_set[q_graph.size][d_graph.size];
7      c_set.filter( q_graph, d_graph,
8          offline_vertex, online_vertex );
9      c_set.compact(d_graph);
10     /* Joining */
11     join_order order(q_graph);
12     list c_edges;
13     c_edges = vertex_join( d_graph,
14         c_set, order );
15     matches m;
16     m = edge_join(c_edges, order);
17 }
```

Fig. 4. Subgraph isomorphism search with G-Morph.

describe the specific algorithms used within each stage. G-Morph uses the *filtering-and-joining* framework with control flow in Fig. 4. Inputs of a query graph $G_q = (V_q, E_q, L_q, l_q)$ and a larger data graph $G_d = (V_d, E_d, L_d, l_d)$, which are compressed sparse row (CSR) undirected graphs, are used to output M containing matches.

The *filtering* phase utilizes additional inputs, the offline & online vertex signature lists $VS\_List(V_d)$ & $VS\_List(V_q)$, respectively. $VS\_List(V_d)$ is generated if null. The 2D array sized $O(|V_q||V_d|)$ stores *true* at index (i, j) if $V_d(j)$ is a candidate of $V_q(i)$ (*false* otherwise). After elimination, this phase generates the *compacted* $C\_Set(V_q)$ (line 9) that stores numerical data for vertex candidates $\forall u \in V_q$.

The *joining* phase first computes a *join order* via BFS of G_q. The first node and subsequent branches are selected according to adopted function $Rank(u) = \frac{freq(L_d(u))}{deg(u)}$ [11], which favors higher degree query vertices with less frequently occurring G_d labels. $C\_Set(V_q)$ is the input in vertex joining that joins adjacent candidate vertices together to form the label-compliant candidate edges (line 13). Lastly, using join order, the candidate edges are *connected* together to build partial M' and final M matches.

4.1 Parallel Filtering

The filtering procedure accepts inputs $VS\_List(V_q)$ and $VS\_List(V_d)$, which are the online and offline vertex signatures, respectively, with the goal of populating $C\_Set(V_q)$, the 2D array that stores candidate vertices. Our vertex signature strategy utilizes LOC and TRI bit-vector encodings described in Sect. 3, which offer fast, branchless, vertex elimination in parallel on the GPU.

A host procedure launches a kernel (see Fig. 5) $\forall u \in V_q$ with $|V_d|$ threads per launch. $C(u) \in C\_Set(V_q)$ is passed to the kernel to set $C(u)$ elements to *true* or *false* (i.e. filtered). Vertices $v \in V_d$ must pass a vertex label filter (line 6) to proceed to

```
1  __device__
2  void kernel_filter(vertex u,
3           graph q_graph, graph d_graph,
4           siglist offline_vertex,
5           siglist online_vertex, set c_set) {
6      if (u.label != d_graph[threadIdx].label
7       || !compare_loc(online_vertex.LOC[u],
8             offline_vertex.LOC[threadIdx])
9       || !compare_tri(online_vertex.TRI[u],
10            offline_vertex.TRI[threadIdx]),
11      || u.degree < d_graph[threadIdx].degree) {
12          c_set[threadIdx] = false;
13     }
14     else {
15          c_set[threadIdx] = true;
16     }
17 }
```

Fig. 5. Parallel filtering kernel.

encoding comparisons (see Sect. 3) to reduce signature memory accesses; comparisons occur in the order of LOC then TRI. Line 11 guarantees that $deg(v)$ is greater than or equal to $deg(u)$, otherwise, v is filtered. A barrier synchronizes kernel launches per u.

4.2 Split-Scan Compaction

The compaction procedure converts the Boolean arrays of $C\_Set(V_q)$ to newly allocated compacted numerical arrays, i.e., each *true* at i is replaced by v_i and each *false* no longer occupies space between numerical values. The algorithm used to generate the output indices array to compact $C\_Set(V_q)$ is *split-scan*, a wrapper for an ordinary GPU-based *scan*. The advantage of split-scan is that it bypasses thread limits per scan.

The split-scan algorithm works as follows on the example in Fig. 6. It first finds the size of each split by dividing the input array size by 2 until the size is less than or equal to a threshold value (e.g. max thread count). The first row of Fig. 6 is the input array, middle rows are intermediate values, and the last row has the output. The split size is 4 (by color); X is a value yet to be accessed. An exclusive *scan* is run on each split iteratively with *adjustments* from each last value (e.g. $0+2$, $1+4$, and $1+5$).

1	0	1	0	0	1	1	1	0	0	0	1	1	1	0	0
0	1	1	2	0	0	1	2	X	X	X	X	X	X	X	X
0	1	1	2	2	2	3	4	0	0	0	0	X	X	X	X
0	1	1	2	2	2	3	4	5	5	5	5	0	1	2	2
0	1	1	2	2	2	3	4	5	5	5	5	6	7	8	8

Fig. 6. Example of split-scan with intermediate values shown.

Compaction iterates each $C(u) \in C\_Set(V_q)$ and uses split-scan to generate the array of indices necessary to compact $C\_Set(V_q)$ sized as the last value of split-scan. Kernels are launched with original and split-scan arrays to populate compacted arrays.

4.3 Slide-Join

Given compacted $C\_Set(V_q)$ and edge-based *join order*, the vertex joining step performs adjacency checks in $C(u_1) \times C(u_2)$ for each edge (u_1, u_2) in the join order. Our sliding window algorithm *slide-join* performs edge verification once and provides overflow protection for larger input via segmentation.

Candidate edges adopt a CSR-like data structure [24]. The candidate edges of a given (u_1, u_2) comprise of arrays: row offsets (abbr. *ver*), column indices (abbr. *edg*), and destination values (abbr. *val*). The last panel (Fig. 7) depicts a final structure.

Slide-join builds CSR-like structures per *join order* edge (u_1, u_2) that represents candidate edges $(v_i, v_j) \in C(u_1) \times C(u_2)$. The Cartesian product size is compared against a max threshold to determine segment usage. It creates the adjacency (abbr. *adj*) array in parallel where the value at $i * |C(u_2)| + j$ is 1 if $v_i \in C(u_1)$ is adjacent to $v_j \in C(u_2)$ s.t. $l_q((u_1, u_2)) = l_d((v_i, v_j))$ (0 otherwise) then compacts *adj* via split-scan. We launch additional kernels to populate each CSR-like array. If the threshold value is exceeded, this process is performed by segment of the $C(u_1)$ against all of $C(u_2)$ with the window size calculated s.t. its value by $|C(u_2)|$ cannot exceed the threshold to build partial structures. Finally, kernels coalesce partials together and reindex *ver* in parallel.

Figure 7 illustrates slide-join with segmented candidate edge building. The window size is 2 and each partial structure is built iteratively. The first panel depicts (v_0, v_1),

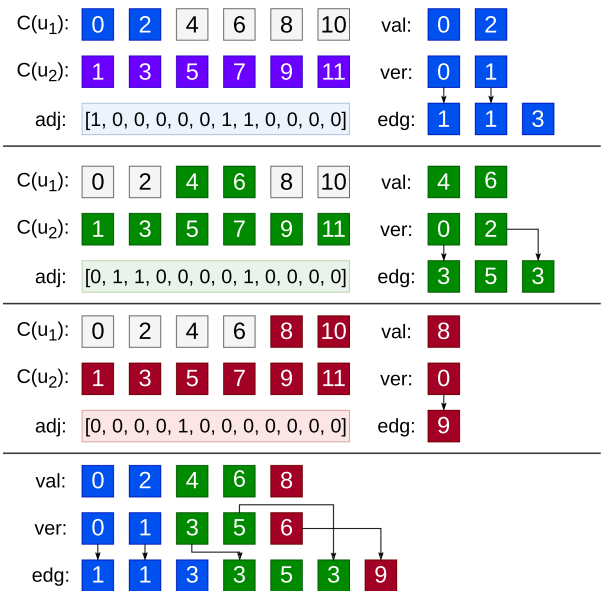


Fig. 7. Depiction of slide-join to join vertices and build candidate edges of a query edge.

(v_2, v_1) and (v_2, v_3) as valid candidate edges from *adj* array using the indexing rules. The last panel depicts the final coalesced structure with reindexed *ver* values.

4.4 Edge Joining

Lastly, we build M from built candidate edges. Iteratively using the *join order*, partial matches M' are built until final matches M are realized. Our adopted edge joining [12,24] strategy counts the number of matches per row in M' then allocates enough memory to write out the matches to M; split-scan determines the write address.

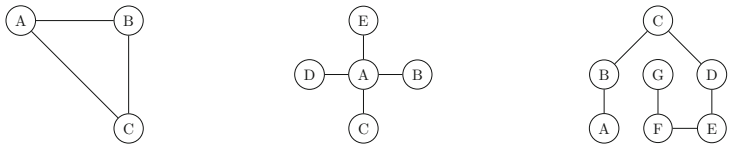
Specifically, this step first builds M' with initially two columns representing the first candidate edge. Subsequent candidate edges are iteratively joined to M', growing column width size, using joining rules between existing M' and the next connecting $C(u_i)$ of the join order, which is done via edge verification kernels with binary search of potential extensions and split-scan for write addresses in M'. Processing continues until partial solutions grow to the final M table after iterating all join order edges.

Furthermore, our system supports both *induced* and *non-induced* matching logic determined by the user and offers split-scan to exploit the sorted candidate edges of slide-join for edge verification. We avoid duplicate checks with guaranteed label matches and use a strided memory layout of M for coalesced memory accesses since each row is accessed by a contiguous thread index.

5 Experimental Evaluation

We evaluate G-Morph with real-world and synthetic graphs by measuring runtimes; scalability is measured by varying query and data sizes. We used some graphs from the Stanford Large Network Dataset Collection (SNAP) [16] repository. Synthetic graphs were generated with PaRMAT [13]. All graphs are undirected with self-loops removed. G-Morph outputs correct, exact solutions and was extensively compared against the output of Boost VF2 [1] and tested systems.

The experiments were performed on a machine with one GPU and one CPU – NVIDIA GeForce RTX 2080 Ti with 68 SMs and 11 GB of GDDR6 RAM, 8-core Intel Core i7-9700K running Ubuntu 18.04, kernel 4.15.0-106-generic, with 32 GB DDR4 RAM and solid-state storage, and CUDA Toolkit 10.2. All runtimes presented are averages over five repetitions of the same experiment. Graph load times and offline metadata generation times on all systems were not factored in and runtimes were calculated when the final solution was stored into memory.



(a) Graph Q_0, triangle (b) Graph Q_1, tree (c) Graph Q_2, path

Fig. 8. Query graphs: Q_0 is needed to evaluate TRI while Q_1 and Q_2 are generic pattern types previously used in other similar evaluations [19].

Table 4. SNAP graph datasets used in evaluation.

	ca-CondMat	com-Amazon	com-DBLP	email-Enron	loc-Brightkite
ID	CM	AM	DB	EN	BK
$\|V_d\|$	23k	335k	317k	37k	58k
$\|E_d\|$	93k	926k	1.05M	184k	214k

	loc-Gowalla	musae-Facebook	web-Google	web-NotreDame	web-Stanford
ID	GO	FB	WG	WN	WS
$\|V_d\|$	197k	22k	876k	326k	282k
$\|E_d\|$	950k	171k	5.11M	1.50M	2.31M

Table 5. Comparison on real-world graphs as runtimes (ms).

Graph	GM-TL	GM-L	GSI	VF3	VF3P
CM	1.04	1.10	8.47	1.35	80
AM	2.80	3.20	44.17	43.16	10,668
DB	3.46	3.82	40.31	41.31	9,491
EN	0.954	0.952	10.39	4.00	132
BK	0.980	0.983	11.74	4.12	269
GO	2.88	3.33	31.65	47.41	3,602
FB	0.89	0.90	9.13	3.45	79.9
WG	19.71	19.93	115.10	385.80	180,628
WN	3.57	3.93	36.60	33.37	6,254
WS	5.38	5.56	54.86	168.17	8,515
WG5	102.27	170.25	118.91	1,633	400,946
WN5	10.85	15.62	38.5	109.21	18,210
WS5	42.46	42.61	71.55	1,827	29,288
EN1	N/A	542.19*	163.32	23,620	N/A
GO1	N/A	9,961*	1,802	167,743	N/A

*LOC off due to unlabeled graph.

5.1 Real-World Graphs

The experiments in this section were designed to measure the effectiveness of our overall strategy for real-world graphs. We obtain runtimes to measure the performance of G-Morph against a variety of real-world SNAP datasets (Table 4). All graphs were undirected; web-type graphs were converted for evaluation. Data graph labeling: $|\mathcal{L}| = 10$ & $|\ell| = 5$ for graphs with no suffix number (e.g. EN), $|\mathcal{L}| = |\ell| = 5$ for graphs suffixed with "5" (e.g. WG5), and $|\mathcal{L}| = |\ell| = 1$ for graphs suffixed with "1" (e.g. EN1). Query graph: Q_0 (Fig. 8(a)) with distinct edge labels (unlabeled in "1" suffixed graphs).

We measure G-Morph (GM) under two modes: (1) GM-TL (LOC & TRI on) and (2) GM-L (LOC on, TRI off). GM is compared against the best existing systems – GPU-based GSI [4] and CPU-based VF3 [3]. The multithreaded version of VF3, VF3P, was

used with 8 threads. Table 5 shows that GM outperforms GSI, VF3, and VF3P for most graphs. GM-TL achieved a max speedup of 15.78× vs. GSI, AM. It obtained speedups of 43.56× vs. VF3, EN1 and 9165× vs. VF3P, WG.

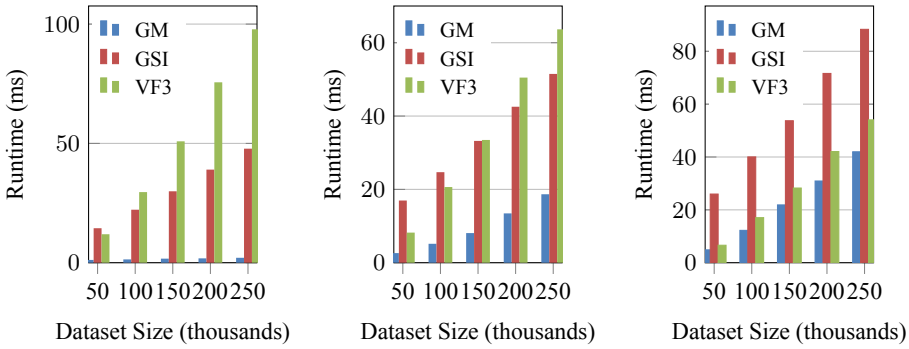
GM-TL outperforms GM-L due to indexing triangles. WG5 experienced the greatest benefit from TRI usage, which proves the effectiveness of TRI especially in larger graphs (WG is the largest tested w.r.t. $|V_d|$). GM-L alone outperforms GSI in most cases, suggesting TRI is optional for smaller graphs if more space is needed since LOC with slide-join often produce ample speedups.

Although this paper's focus is labeled graphs, two unlabeled experiments (EN1 & GO1) were run for completeness with LOC & TRI off. GM easily outperformed VF3 in these unlabeled experiments, but GSI outperformed GM in this case. VF3P crashed with no results here and VF3P also experienced relatively slow runtimes throughout.

5.2 Scalability

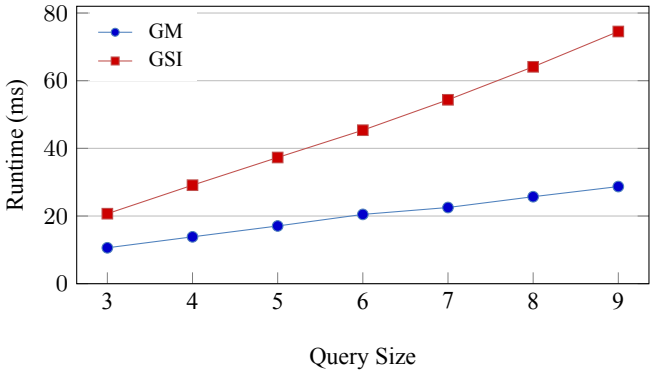
This subsection measures G-Morph performance against other systems by increasing the data graph and query graph size, respectively. All experiments have LOC enabled while the experiment using Q_0 also has TRI enabled.

Data Graph Size: We study the scalability of G-Morph with varying *data graph* size. Five input RMAT graphs started at size $|V_d| = 50$k and $|E_d| = 200$k and doubly scaled $|V_d|$ & $|E_d|$ (largest $|V_d| = 250$k and $|E_d| = 1$M). The same datasets were run vs. three query patterns (Fig. 8). The labeling scheme was: Q_0 as $|\mathcal{L}| = |\ell| = 3$, Q_1 as $|\mathcal{L}| = |\ell| = 5$, and Q_2 as $|\mathcal{L}| = 7$ & $|\ell| = 3$. G_d used the same labeling counts per query graph. G-Morph (GM) is compared against GSI and VF3 with induced matches.

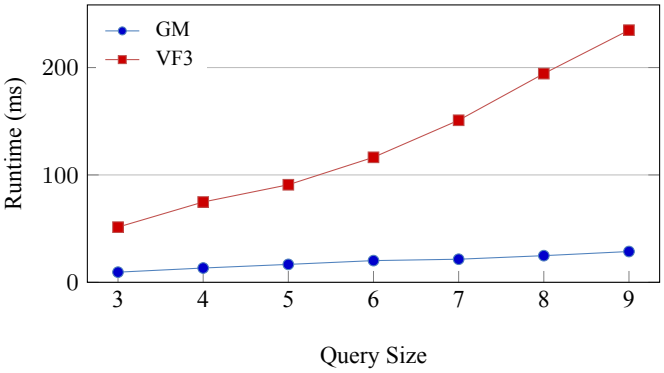


(a) Triangle Q_0 as Input. (b) Tree Q_1 as Input. (c) Path Q_2 as Input.

Fig. 9. Scalability with increasing data size.



(a) **Non-induced**, r-100k



(b) **Induced**, r-100k

Fig. 10. Scalability with increasing query size.

Figure 9 shows results across all query graphs from Fig. 8 with runtimes (ms) against dataset sizes. GM outperforms the best existing systems with promising scalability. GM speedups are significant against smaller datasets but level out as they scale. GM achieves speedups up to 23.15×, 6.47×, and 5.34× vs. GSI on respective patterns. Speedups vs. VF3 were around 47.39×, 4.14×, and 1.39× on respective patterns.

Query Graph Size: We measure GM scalability with varying *query* size. Path queries with an incrementing size were used against an RMAT dataset ($|V_d| = 100$k, $|E_d| =$ 2M, $|\mathcal{L}| = 10$, $|\ell| = 5$). V_q were distinctly labeled; E_q labeled with incrementing value $\mathrm{mod}|\ell|$. Experimental configuration: *induced* mode *off* vs. GSI and *on* vs. VF3.

GM outperformed both GSI and VF3 in these experiments too. Figure 10 shows promising scalability of GM against GSI and especially VF3. GM's runtime increases very slowly with query size, especially against VF3 with *induced* matches. GM's speedups against GSI and VF3 were 2.6× and 8.19×, respectively, for the query size of 9.

6 Conclusions

We presented G-Morph, an efficient GPU-based subgraph isomorphism search system on labeled graphs. We proposed a novel space-efficient vertex signature strategy that can be implemented as integers with good filtering power of proposed LOC and TRI codes implementing *local+cycle indexing*. By reducing downstream Cartesian products and improving edge verification, extraneous computations are limited. We also proposed a novel joining procedure *slide-join*, a sliding window algorithm that avoids duplicate edge verification and offers overflow protection; *split-scan* handles large scans. Experiments on labeled real-world graphs show G-Morph outperforming both GSI and VF3.

Acknowledgement. This work is supported in part by National Science Foundation grants CCF-1813173, CCF-2002554, and CCF-2028714 to the University of California Riverside.

References

1. Boost Graph Library: VF2 (Sub)Graph Isomorphism - master (2020). https://www.boost.org/doc/libs/master/libs/graph/doc/vf2_sub_graph_iso.html. Accessed July 2020
2. Chapter 39. Parallel Prefix Sum (Scan) with CUDA (2020). https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda. Accessed July 2020
3. MiviaLab/vf3lib: VF3 Algorithm - The fastest algorithm to solve subgraph isomorphism on large and dense graphs (2020). https://github.com/MiviaLab/vf3lib. Accessed July 2020
4. bookug/GSI: GPU-friendly subgraph isomorphism (2020). https://github.com/bookug/GSI. Accessed Oct 2020
5. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinformatics **14**(S7), S13 (2013)
6. Carletti, V., Foggia, P., Saggese, A., Vento, M.: Introducing VF3: a new algorithm for subgraph isomorphism. In: Foggia, P., Liu, C.-L., Vento, M. (eds.) GbRPR 2017. LNCS, vol. 10310, pp. 128–139. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58961-9_12
7. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)
8. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. In: Workshop on Graph-Based Representations in Pattern Recognition, pp. 149–159 (2001)
9. Fan, W., Wang, X., Wu, Y.: Diversified top-k graph pattern matching. Proc. VLDB Endowment **6**(13), 1510–1521 (2013)
10. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Workshop on General Purpose Processing on Graphics Processing Units, pp. 1–8 (2011)
11. Han, W.S., Lee, J., Lee, J.H.: TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: ACM SIGMOD International Conference on Management of Data, pp. 337–348 (2013)
12. He, B., et al.: Relational joins on graphics processors. In: ACM SIGMOD International Conference on Management of Data, pp. 511–524 (2008)
13. Khorasani, F., Gupta, R., Bhuyan, L.N.: Scalable SIMD-efficient graph processing on GPUs. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 39–50 (2015)

14. Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N.: CuSha: vertex-centric graph processing on GPUs. In: International Symposium on High-performance Parallel and Distributed Computing, pp. 239–252 (2014)
15. Lee, J., Han, W.S., Kasperovics, R., Lee, J.H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proc. VLDB Endowment **6**(2), 133–144 (2012)
16. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data
17. McCreesh, C., Prosser, P., Solnon, C., Trimble, J.: When subgraph isomorphism is really hard, and why this matters for graph databases. J. Artif. Intell. Res. **61**, 723–759 (2018)
18. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput. Aided Mol. Des. **16**(7), 521–533 (2002)
19. Reza, T., Ripeanu, M., Tripoul, N., Sanders, G., Pearce, R.: PruneJuice: pruning trillion-edge graphs to a precise pattern-matching solution. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 265–281 (2018)
20. Serafini, M., De Francisci Morales, G., Siganos, G.: QFrag: distributed graph search via subgraph isomorphism. In: Symposium on Cloud Computing, pp. 214–228 (2017)
21. Shamir, R., Tsur, D.: Faster subtree isomorphism. In: Israeli Symposium on Theory of Computing and Systems, pp. 126–131 (1997)
22. Shi, C., Li, Y., Zhang, J., Sun, Y., Philip, S.Y.: A survey of heterogeneous information network analysis. IEEE Trans. Knowl. Data Eng. **29**(1), 17–37 (2016)
23. Song, Q., Wu, Y., Lin, P., Dong, L.X., Sun, H.: Mining summaries for knowledge graph search. IEEE Trans. Knowl. Data Eng. **30**(10), 1887–1900 (2018)
24. Tran, H.-N., Kim, J., He, B.: Fast subgraph matching on large graphs using graphics processors. In: Renz, M., Shahabi, C., Zhou, X., Cheema, M.A. (eds.) DASFAA 2015. LNCS, vol. 9049, pp. 299–315. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18120-2_18
25. Ullmann, J.R.: An algorithm for subgraph isomorphism. JACM **23**(1), 31–42 (1976)
26. Wang, L., Wang, Y., Owens, J.D.: Fast parallel subgraph matching on the GPU. In: High Performance Parallel and Dist. Computing (2016)
27. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: a high-performance graph processing library on the GPU. In: Symposium on Principles and Practice of Parallel Programming, pp. 1–12 (2016)
28. Webber, J.: A programmatic introduction to Neo4j. In: Conference on Systems, Programming, and Apps: Software for Humanity, pp. 217–218 (2012)
29. Zeng, L., Zou, L., Özsu, M.T., Hu, L., Zhang, F.: GSI: GPU-friendly subgraph isomorphism. In: International Conference on Data Engineering, pp. 1249–1260 (2020)
30. Zhang, S., Li, S., Yang, J.: GADDI: distance index based subgraph matching in biological networks. In: International Conference on Extending Database Technology: Advances in Database Technology, pp. 192–203 (2009)
31. Zheng, W., Zou, L., Lian, X., Hong, L., Zhao, D.: Efficient subgraph skyline search over large graphs. In: ACM International Conference on Information and Knowledge Management, pp. 1529–1538 (2014)