SMOG: Accelerating Subgraph Matching on GPUs

Zhibin Wang[1,2], Ziheng Meng[1], Xue Li[2], Xi Lin[1], Long Zheng[3,4], Chen Tian[1], Sheng Zhong[1]
State Key Laboratory for Novel Software Technology, Nanjing University[1], Alibaba Group[2],
National Engineering Research Center for Big Data Technology and System/
Services Computing Technology and System Lab/Cluster and Grid Computing Laboratory,
Huazhong University of Science and Technology[3], Zhejiang Lab[4]
wzbwangzhibin@gmail.com, jshamzh@gmail.com, youli.lx@alibaba-inc.com, 350904583lx@gmail.com,
longzh@hust.edu.cn, tianchen@nju.edu.cn, sheng.zhong@gmail.com

***Abstract*—Subgraph matching is a crucial problem in graph theory with diverse applications in fields, such as bioinformatics, social networks and recommendation systems. Accelerating subgraph matching can be greatly facilitated by GPUs, which offer exceptional parallelism and high memory bandwidth. By leveraging the power of multiple GPU cards, subgraph matching can be scaled to achieve unprecedented levels of performance.**

In this paper, we propose SMOG, an abbreviation for Subgraph Matching On Multi-Card GPUs. It is a *general*, *high-performance* and *scalable* subgraph matching system that utilizes multi-card GPUs. To address the issue of duplication resulting from subgraph automorphism, SMOG introduces a two-step approach. Firstly, it analyzes the symmetry within the subgraph. Then, it adaptively adjusts the graph preprocessing and generates subgraph-aware GPU codes tailored to the given subgraph. Furthermore, SMOG leverages multi-level parallelism by designing the specific strategy for each level, enabling it to scale from 1 to 1,024 GPU cards, resulting in an extraordinary $553\times$ speedup.

We evaluate SMOG on various subgraph queries and datasets. The experimental results demonstrate that SMOG outperforms the triangle-specific system TRUST with an average speedup of $2.94\times$. And it performs significantly better than the subgraph matching system RPS by $203.55\times$ and the graph processing system Gunrock by $35,455.52\times$ on average.

***Index Terms*—Graph, GPU, Subgraph matching**

I. Introduction

Subgraph matching, which aims to find all mappings of a given subgraph (such as a triangle) in a data graph, has a broad spectrum of applications including community detection [1], [2], [3], [4], biological/chemistry analysis [5], [6], [7], [8] and emerging graph neural networks [9], [10], [11]. The GraphChallenge [12] competition also has established a subgraph isomorphism challenge for the subgraph matching problem. In GraphChallenge, there have been plenty of works that focus on accelerating triangle counting [13], [14], [15], [16], [17], [18], while [19] further considers square counting. Instead of only counting triangles/squares, subgraph matching that supports a variety of queried subgraphs has a wider range of applications and is much more complex, making it a worthwhile area of study. However, with the increasing size of modern graphs, subgraph matching poses significant computational challenges due to its NP-complete complexity [20]. As indicated by the 2022 innovation award winner FAST [21], subgraph matching systems perform significantly worse than triangle counting systems on the triangle counting task.

Fortunately, the acceleration of subgraph matching can be greatly facilitated by GPUs, which offer massive parallelism and high memory bandwidth. However, fully leveraging these capabilities to design a system for efficient subgraph matching on GPUs is non-trivial due to the following challenges.

Challenge 1: How to eliminate duplicated enumeration. Duplication occurs when multiple mappings of a subgraph to a set of vertices in the graph are counted, leading to increased overhead in subgraph matching. As indicated in [22], [23], there are six mappings between two triangles, resulting from different permutations of the vertices in the triangle.

To tackle this challenge, we investigate the reason for duplication, which is automorphism caused by symmetry in the subgraph. Accordingly, we leverage the symmetry-breaking technique [24], [25] by setting restrictions on the mapping. Observing that restrictions can be classified into two types according to the characteristics of the input subgraph, we propose an adaptive solution allowing for the efficient elimination of duplication based on the properties of the ordered set of vertices in the mapping. For a totally (linearly) ordered set, such as a triangle, we leverage the orientation optimization, widely used in GraphChallenge's triangle counting systems [13], [14], [16], [17], during preprocessing. For a partially ordered set, such as a 4-cycle, we adopt the technique of intersection with restrictions [24], [25], involving incorporating restrictions on the candidates after the intersection in the runtime.

Challenge 2: How to leverage massive parallelism provided by multi-card GPUs. Modern GPUs are designed to provide massive parallelism, while multi-card GPUs further increase the parallelism to an unprecedented degree. However, effectively utilizing this parallelism requires careful design to avoid idle threads and workload imbalance [26], [13], [27].

To leverage the massive parallelism offered by multi-card GPUs, we decompose the parallelism into three levels, including GPU card level parallelism, warp level parallelism and thread level parallelism, and we design strategies for each level of parallelism. Specifically, we propose 1) workload partitioning for GPU card level parallelism, optimizing the partitioning of tasks among multiple GPU cards; 2) task scheduling for warp level parallelism, efficiently managing the execution of tasks among warps; and 3) hash-based intersection for thread level parallelism, utilizing the threads within a warp to accelerate the intersection operation.

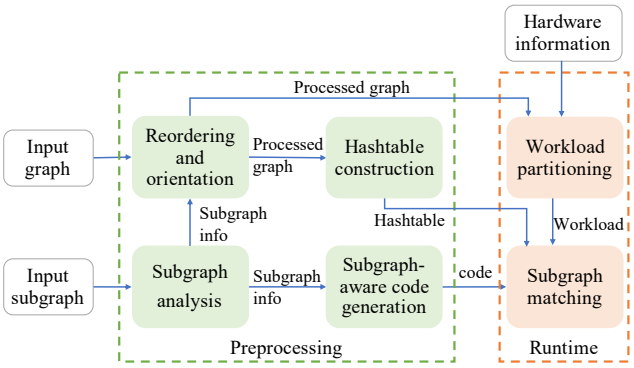


Fig. 1: System overview.

As a result, we propose SMOG, short for Subgraph Matching On Multi-Card GPUs. It is a *general*, *high-performance* and *scalable* subgraph matching system that utilizes multi-card GPUs. Figure 1 illustrates the workflow of SMOG, which comprises two key components: preprocessing and runtime. In the preprocessing stage, the input subgraph is first analyzed to obtain its characteristics for the elimination of duplication (Section III). This information not only determines the preprocessing strategy for the data graph, including reordering and orientation, but also guides the subgraph-aware code generation for the runtime. Once the preprocessing of the data graph is completed, the processed graph is fed into the hashtable construction procedure to build the corresponding hashtables following the methodology outlined in [27]. When both the generated code and hashtables are ready, the runtime of the system is initiated. The runtime stage leverages the multi-level parallelism of GPUs to accelerate subgraph matching (Section IV). Specifically, the runtime includes a workload partitioning step that distributes the workload according to the hardware information. After that, the subgraph matching kernel is launched and executed to perform the subgraph matching algorithm (Section II).

We summarize the contributions of this paper as follows:

- The design of an adaptive strategy that dynamically adjusts the search space based on analyzing the characteristics of the subgraph.
- The exploration and utilization of multi-card GPUs by decomposing the massive parallelism into multiple levels and designing approaches for workload partitioning, task scheduling and hash-based intersection.
- The proposal of the general subgraph matching framework - SMOG, which combines innovative techniques to address the challenges of duplicated enumeration and massive parallelism exploitation.
- Extensive experimentation and evaluation on various subgraph matching problems and a range of datasets. The experimental results demonstrate that SMOG not only significantly outperforms state-of-the-art subgraph matching system RPS and graph processing system Gunrock by an average of $203.55\times$ and $35,455.52\times$, respectively, but also beats the specific triangle counting system TRUST by an average factor of $2.94\times$ on the triangle counting problem. Furthermore, SMOG exhibits remarkable scalability, achieving a speedup of up to $553\times$ when scaling from 1 to 1,024 GPU cards.

II. Subgraph Matching Algorithm

A. Definition

We denote $G = (V, E)$ ($\mathcal{G} = (\mathcal{V}, \mathcal{E})$) as a graph (subgraph) with a vertex set and an edge set. A vertex (edge) in the graph (subgraph) is then denoted by $v \in V$ ($e \in E$) and $u \in \mathcal{V}$ ($\epsilon \in \mathcal{E}$), respectively. The neighbors (degree) of a given vertex v are represented by $N(v)$ ($d(v) = |N(v)|$). Additionally, the cardinality or size of a set/array X is denoted by $|X|$.

Subgraph matching/isomorphism: Given G and $\mathcal{G}$, a mapping $\mathcal{M}$ maps a vertex of subgraph $u_i \in \mathcal{V}$ to a vertex of graph $v = \mathcal{M}(u_i) \in V$, ensuring that if an edge exists in the subgraph $(u_i, u_j) \in \mathcal{E}$, there must also be an edge in the graph $(\mathcal{M}(u_i), \mathcal{M}(u_j)) \in E$. In GraphChallenge, the subgraph isomorphism problem asks whether there exists a mapping between G and $\mathcal{G}$. However, the subgraph matching task is more complex as it aims to find the set of all mapping results M, where each mapping $\mathcal{M} \in M$. For simplicity, we use m_i to denote $\mathcal{M}(u_i)$ and maintain $\mathcal{M}$ as an array $[m_0, \cdots, m_{|\mathcal{V}|-1}]$. Moreover, we consider that there is a fixed vertex order in the subgraph $\mathcal{O} = [u_0, \cdots, u_{|\mathcal{V}|-1}]$, and we follow this order to conduct the matching.

B. Algorithm

Algorithm 1 Subgraph matching

Input: Subgraph $\mathcal{G}$, matching order $\mathcal{O} = [u_0, \cdots, u_{|\mathcal{V}|}]$, graph G.
Output: Mapping set M

```
1:  for m_0 ∈ V do in parallel
2:      Initialize M ← [m_0]
3:      DFS(1, M)
4:  function DFS(i, M)
5:      if |M| = |V| then
6:          M ← M ∪ {M}
7:          return
8:      S ← {j | j < i and (u_i, u_j) ∈ E}.
9:      Candidate C ← N(m_k), where k = arg min_{k∈S}(d(m_k))
10:     for j ∈ S and j ≠ k do
11:         C ← C ∩ N(m_j)
12:     for m_i ∈ C do
13:         DFS(i + 1, [M, [m_i]])
```

There are two main-stream subgraph matching algorithms: BFS-based [28], [29], [30] and DFS-based [25], [31]. The BFS-based approach enumerates the mapping level by level, maintaining partial mappings in memory for each level. However, the limited GPU memory makes it impractical for super-linear intermediate results. Thus, SMOG adopts a DFS-based approach that has controllable memory usage by enumerating mappings one by one.

Algorithm 1 describes the subgraph matching algorithm of SMOG. Naturally, the matching tasks starting from different vertices are independent of each other and can be parallelized (line 1). The DFS function (line 4) is used to find the mapping m_i of the i-th vertex u_i in the subgraph. Particularly, lines 5-7 check if one mapping is complete. To find the mapping of the next vertex u_i, given that i vertices have already been matched, the search extended from the matched vertex u_j connected to u_i. Accordingly, we maintain the index of u_j in S (line 8). The mapping vertex m_i will fall into the intersection of neighbors of m_j, i.e., $\bigcap_{j \in S} N(m_j)$ (lines 9-11). To minimize the time complexity, we start from the minimal neighbor list (line 9). After obtaining the candidate set $\mathcal{C}$ of mapping m_i, we proceed to match the next vertex (line 13).

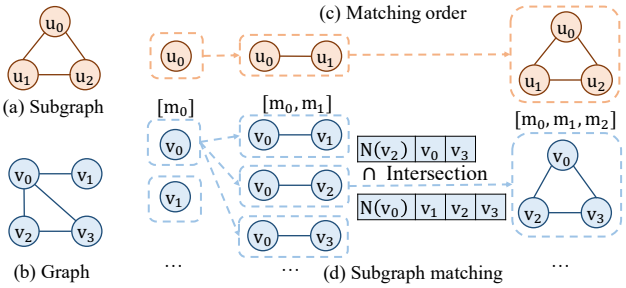


Fig. 2: An example of subgraph matching.

Figure 2 depicts the subgraph matching when the subgraph is a triangle (Figure 2(a)). The data graph is shown in Figure 2(b). We follow the order $\mathcal{O} = [u_0, u_1, u_2]$ to match the subgraph (Figure 2(c)). Figure 2(d) demonstrates how to match the subgraph by extending the mapping one vertex at a time. At first, we start from each vertex in the graph to find the mapping of u_0. For a mapping $m_0 = v_0$, we obtain m_1 by enumerating the neighbors of v_0, i.e., v_1, v_2, v_3. Next, considering a neighbor v_2 and the corresponding mapping $[m_0, m_1] = [v_0, v_2]$, we intersect the neighbors of v_0 and the neighbors of v_2 to obtain the candidates of m_3. Finally, a mapping $[m_0, m_1, m_2] = [v_0, v_2, v_3]$ is enumerated.

III. Elimination of Duplication

Considering the example in Figure 2, there are a total of $A_3^3 = 6$ mappings between triangle (v_0, v_2, v_3) and (u_0, u_1, u_2). This count arises from the fact that each permutation of the vertices $\{v_0, v_2, v_3\}$ in the graph can be mapped to the given triangle. The duplication of mapping significantly increases the overhead of subgraph matching.

In this section, we first investigate the reason for duplication. Then we leverage a symmetry-breaking technique, which sets restrictions on the mapping to eliminate duplication. Recognizing two distinct categories of these restrictions, each benefiting from different optimization, we propose an adaptive solution depending on input subgraphs for optimal performance.

A. Elimination of Duplication by Symmetry Breaking

Reason for duplication: Recall the triangle matching example, the subgraph (u_0, u_1, u_2) can be mapped to any permutation of itself, namely automorphism. Accordingly, duplication appears. Moreover, as indicated in [32], the automorphism is implicit in the symmetry in the subgraph.

Symmetry breaking: The most straightforward way to eliminate the duplication is by breaking the symmetry [24], [25]. In triangle matching, vertices u_0 and u_1 are symmetry, i.e., swapping them still results in a triangle. Actually, we can break this symmetry by setting a restriction on the mapping of u_0 and u_1, i.e., $p(m_0) > p(m_1)$, where $p(m)$ is the unique priority of the vertex m in the graph. Obviously, there may exist more than one symmetry/restriction, and we identify the symmetry of a vertex based on its matching order. Specifically, for matching the i-th vertex, we first label the matched vertices with their IDs and treat the remaining vertices as unlabeled. We then identify the symmetry group SG containing vertex i in the unlabeled vertices and set the restrictions between i and $j \in SG - \{i\}$. Consider the examples in Figure 3:

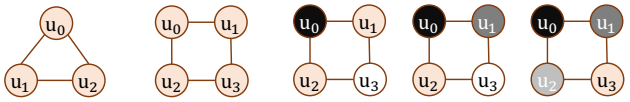


Fig. 3: Symmetry breaking.

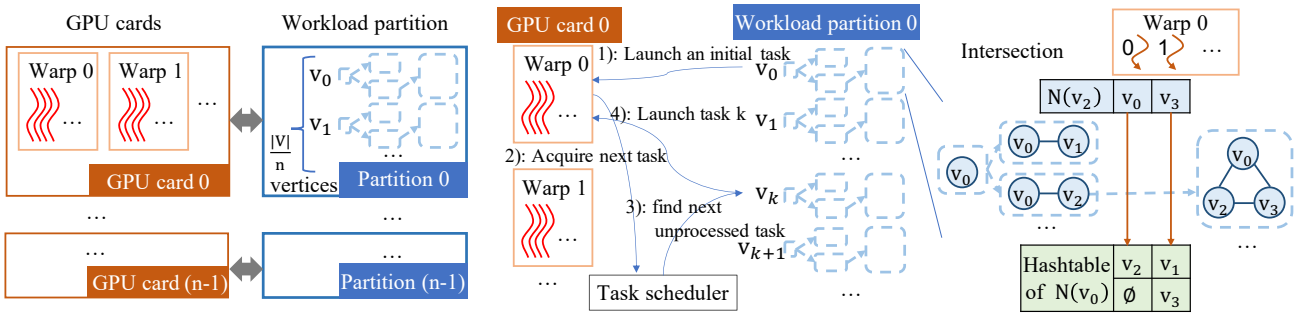
Triangle: In the case of a triangle, u_0 is symmetrical to both u_1 and u_2, indicating the restrictions $p(m_0) > p(m_1)$ and $p(m_0) > p(m_2)$. After u_0 is matched, since the remaining two vertices are still symmetrical, $p(m_1) > p(m_2)$ is set. Overall, a total order is formed as $p(m_0) > p(m_1) > p(m_2)$.

4-cycle: A 4-cycle (a.k.a. rectangle, or butterfly) contains four vertices and four edges, defined as (u_0, u_1, u_2, u_3). Figure 3(c) depicts the symmetry in a 4-cycle. In the beginning, the four vertices are symmetrical, implying restrictions $p(m_0) > p(m_1)$, $p(m_0) > p(m_2)$ and $p(m_0) > p(m_3)$. Next, after u_0 is matched, we find that u_1 and u_2 are symmetrical as they are both connected to u_0 and u_3. Accordingly, we add a restriction $p(m_1) > p(m_2)$. For the remaining vertices, u_2 and u_3, no more restrictions are required due to the lack of symmetry. Overall, a partial order is formed as $p(m_0) > p(m_1) > p(m_2), p(m_0) > p(m_3)$.

B. Adaptive Strategy for Elimination

The restrictions make the set of vertices in the mapping become an ordered set, which can be classified into the following two categories, each with its corresponding strategies to eliminate duplication:

Totally (linearly) ordered set (orientation in preprocessing): Regarding a triangle, restrictions exist among all pairs of vertices, forming a totally ordered set. It provides an excellent property that the restrictions can be concluded by a linear ordered list such as $p(m_0) > p(m_1) > p(m_2)$. Clearly, cliques of any size satisfy the total order due to perfect symmetry, i.e., symmetry exists between each pair of vertices. The linear property of the totally ordered set suggests that for two vertices in mapping m_i, m_j satisfying $j < i$, $p(m_j) > p(m_i)$. Referring to lines 8-11 in Algorithm 1, the i-th vertex is extended from the previous mapping vertex j. Combining the linear property and matching process, we observe that



(a) Distribute the workload to GPU cards (b) Schedule the workload among the warps (c) Accelerate intersection by a warp

Fig. 4: Accelerate subgraph matching by multiple GPU cards.

only one direction of an edge is required in the matching. Specifically, for a edge (v_x, v_y), where $p(v_x) > p(v_y)$, we will only traversal from high-priority vertex v_x to low-priority vertex v_y, as v_x is always in front of v_y in the matching process. Accordingly, we preprocess the graph by orienting the edges based on the priorities, namely orientation, which is also widely discussed in triangle counting [22], [23].

Partially ordered set (intersection with restrictions): For other subgraphs, such as the 4-cycle, we call its vertices set a partially ordered set. The orientation optimization does not apply to these general subgraphs. In these cases, discovering high-priority vertices from low-priority vertices is necessary. For example, in a 4-cycle (u_0, u_1, u_2, u_3), finding candidates for m_3 requires intersecting the neighbor lists of m_1 and m_2. As there does not exist priority restriction between m_1 and m_3, discovering m_3 from m_1 with higher priority is possible.

To address this limitation, we adopt the technique of intersection with restrictions [24], [25], which further checks the restrictions for candidates generated by intersection. These restrictions are incorporated into the algorithm as an additional input parameter $\mathcal{R} = [r_0, \cdots, r_{|\mathcal{V}|}]$, where r_i represents the restrictions for matching u_i, indicating that the priority of m_i is lower than some other matched vertices. Referring back to the previous 4-cycle example, the restrictions would be $r_0 = \emptyset$, $r_1 = \{0\}$, $r_2 = \{1\}$ and $r_3 = \{0\}$.

Set the priority by reordering: As indicated in [22], [33], [24], giving low priority to a vertex with a high degree can limit the search tree of this vertex and thus improve the performance. In practice, we reorder the vertices by degree and utilize the new index of the vertex as its priority.

Adaptive preprocessing and matching algorithm: The two strategies with different runtime and preprocessing operations mentioned above are complementary to each other. The orientation optimization is more efficient but only applicable for cliques, while the intersection with restrictions is applicable for general subgraphs but less efficient. As a result, we adopt an adaptive approach as shown in Figure 1. We first analyze the subgraph and then dispatch different preprocessing steps as well as generate different GPU kernel codes accordingly.

IV. Accelerating Subgraph Matching via Multiple GPU Cards

GPUs are well-suited for accelerating subgraph matching due to their massive parallelism and high throughput bandwidth. Additionally, by utilizing multiple GPU cards, the scalability of subgraph matching can be further enhanced. To fully exploit the massive parallelism provided by multiple GPU cards, we decompose the parallelism into three levels, including GPU card level parallelism, warp level parallelism and thread level parallelism. The architecture diagram presented in Figure 4 showcases the multi-level parallelism in our system, encompassing three crucial procedures: workload partitioning, inter-warp scheduling and intersection. In this section, we follow a top-down fashion to describe our design.

A. Workload Partitioning for Multiple GPU Cards

Firstly, we address GPU card level parallelism. As shown in Figure 4(a), the workload partitioning procedure evenly distributes the vertices in the preprocessed graph into n partitions for n GPU cards, with each partition assigned to a GPU card. The vertices in a partition represent the starting vertices of the matching process (line 1 of Algorithm 1), which are further scheduled to the warps on the GPU card.

B. Inter-warp Task Scheduling

Next, we present the approach for warp level parallelism within a GPU card. As GPUs follow the single instruction multiple threads (SIMT) model, where 32 threads in a warp execute the same instruction simultaneously, we assign a warp to handle the subgraph matching task starting from a given vertex (lines 2-3 of Algorithm 1). Figure 4(b) demonstrates the task scheduling among warps within a GPU card. In more detail, each warp is first assigned a vertex to start the process of subgraph matching. When the current task is finished, the warp requests the task scheduler for a new vertex and launches the corresponding next task. This procedure is repeated until all vertices in the partition are processed.

C. Leveraging Warps for Accelerating Intersection

Finally, we consider the thread level parallelism within a warp. Regarding the DFS function of Algorithm 1, the preparation steps (lines 8-9) have negligible cost, thus allowing all threads in a warp to perform the same operation to obtain the

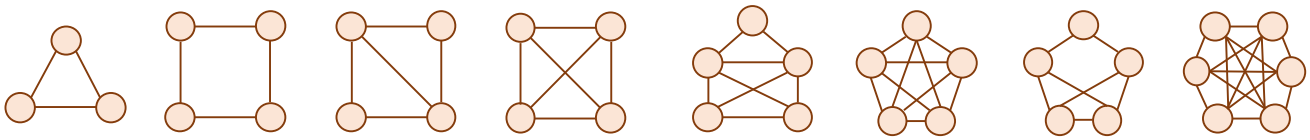


Fig. 5: Queried subgraphs.

TABLE I: Runtime of SMOG, TRUST, RPS and Gunrock on triangle counting task (in ms).

Datesets	Vertices	Edges	Triangles	SMOG	TRUST	Speedup	RPS	Speedup	Gunrock	Speedup
Real-world graphs from SNAP (Stanford's Large Network Dataset Collection)										
amazon0312	400,727	2,349,869	3,686,467	1.53	0.69	0.45	8.09	5.29	9563.66	6250.76
amazon0505	410,236	2,439,437	3,951,063	1.91	0.69	0.36	8.82	4.61	10249.27	5363.30
amazon0601	403,394	2,443,408	3,986,507	2.21	0.77	0.35	8.97	4.05	10167.15	4594.28
cit-Patents	3,774,768	16,518,947	7,515,023	16.02	3.13	0.20	53.72	3.35	#OT	#OT
flickrEdges	105,718	2,081,520	107,987,357	4.43	8.18	1.85	30.13	6.80	1932.72	436.28
roadNet-CA	1,965,206	2,766,607	120,676	2.26	0.61	0.27	11.91	5.27	55708.25	24638.77
roadNet-PA	1,088,092	1,541,898	67,150	0.76	0.28	0.37	8.59	11.33	17336.56	22871.45
roadNet-TX	1,379,917	1,921,660	82,869	1.26	0.35	0.28	9.84	7.80	26823.86	21255.04
friendster	65,608,366	1,806,067,135	4,173,724,142	4770.00	1469.83	0.31	#OT	#OT	#OT	#OT
Synthetic Kronecker Graphs (Theory) with many/some triangles										
25-81-256-B1k	547,924	2,132,284	2,102,761	1.01	1.58	1.56	4.90	4.86	132.95	131.89
25-81-256-B2k	547,924	2,132,284	7	0.54	0.98	1.82	30.85	57.45	12.28	22.87
3-4-5-9-16-25-B1k	530,400	11,080,030	35,882,427	17.38	68.38	3.94	156.66	9.02	74165.40	4268.51
3-4-5-9-16-25-B2k	530,400	11,080,030	651	5.59	16.50	2.95	104.24	18.65	66878.74	11966.14
4-5-9-16-25-B1k	132,600	1,582,861	3,548,463	1.37	1.69	1.23	26.96	19.72	161023.54	117793.38
4-5-9-16-25-B2k	132,600	1,582,861	155	0.63	0.91	1.43	16.89	26.77	2294.29	3635.96
5-9-16-25-81-B1k	2,174,640	28,667,380	66,758,995	31.77	569.74	17.94	610.24	19.21	#OT	#OT
5-9-16-25-81-B2k	2,174,640	28,667,380	155	11.58	130.94	11.31	370.95	32.04	485469.40	41937.58
9-16-25-81-B1k	362,440	2,606,125	4,059,175	2.79	4.54	1.63	50.64	18.18	71766.79	25769.05
9-16-25-81-B2k	362,440	2,606,125	35	1.16	2.59	2.24	37.71	32.59	5153.34	4454.05
Protein k-mer graphs generated using data from GenBank										
P1a	139,353,211	148,914,992	3,412	77.01	72.85	0.95	446.91	5.80	#OT	#OT
U1a	67,716,231	69,389,281	325	35.20	33.86	0.96	258.96	7.36	#OT	#OT
V1r	214,005,017	232,705,452	49	116.60	100.22	0.86	578.91	4.97	#OT	#OT
V2a	55,042,369	58,608,800	1,443	30.30	17.11	0.56	209.83	6.93	#OT	#OT
Graphs from MAWI Working Group Traffic Archive										
201512012345	18,571,154	19,020,160	2	0.57	5.22	9.10	741.92	1292.54	58753.08	102357.28
201512020000	35,991,342	37,242,710	2	1.13	10.15	8.97	2540.68	2246.40	151281.38	133758.96
201512020030	68,863,315	71,707,480	6	2.72	19.39	7.13	4776.98	1756.24	354289.14	130253.36
201512020130	82,389,971	89,009,590	10	6.58	37.27	5.67	8348.55	1269.74	735044.34	111793.82
Synthetic graph500 network										
scale18-ef16	174,147	3,800,348	82,287,285	7.20	9.58	1.33	65.14	9.05	114594.95	15920.39
scale19-ef16	335,318	7,729,675	186,288,972	20.36	26.89	1.32	159.38	7.83	408176.95	20047.98
scale20-ef16	645,820	15,680,861	419,349,784	57.16	85.08	1.49	315.34	5.52	1669641.48	29208.94
scale21-ef16	1,243,072	31,731,650	935,100,883	153.25	273.14	1.78	764.00	4.99	#OT	#OT
scale22-ef16	2,393,285	64,097,004	2,067,392,370	423.08	857.73	2.03	1807.42	4.27	#OT	#OT
scale23-ef16	4,606,314	129,250,705	4,549,133,002	1109.78	2877.24	2.59	4474.52	4.03	#OT	#OT
scale24-ef16	8,860,450	260,261,843	9,936,161,560	2883.68	9425.55	3.27	11233.00	3.90	#OT	#OT
scale25-ef16	17,043,780	523,467,448	21,575,375,802	6786.56	29935.40	4.41	28424.40	4.19	#OT	#OT

lists that need to be intersected. For the intersection operation, we follow [27] by leveraging a hash-based intersection, with all 32 threads in a warp being utilized. This is done by searching for the vertices of the list in the hashtable in parallel.

V. Evaluation

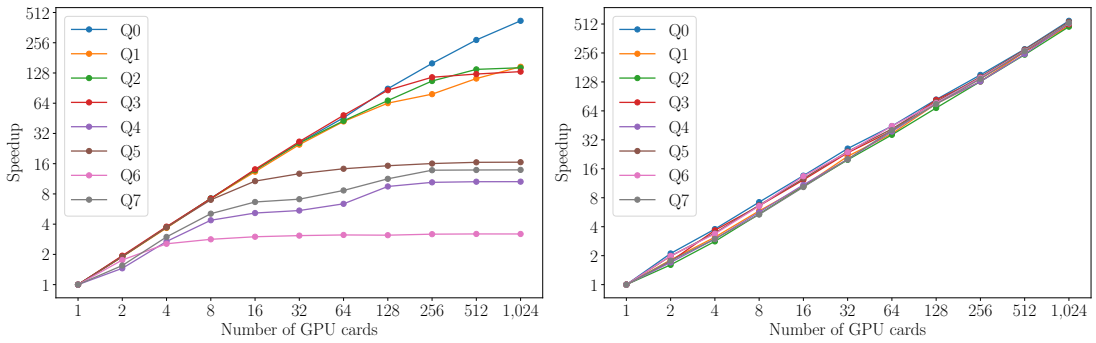
SMOG[1] is implemented with C++/CUDA code for subgraph matching and Python for code generation. We evaluate SMOG on A100 [34] GPUs, each of which has 40 GB GPU memory. The datasets used in our experiments are sourced from the official website of the Graph Challenge [12], including SNAP datasets [35], synthetic kronecker graphs (theory and graph500) [36], protein k-mer graphs [37] and MAWI graphs [38]. For our experiments, besides the triangle, which is required by the GraphChallenge, we also evaluate several other subgraphs, as listed in Figure 5[2]. These queried subgraphs have been evaluated in several recent works [30], [39]. We follow H-index [13] to measure the runtime of SMOG, i.e., reporting the GPU kernel time once the graph is loaded on GPUs. For the experiment of scalability, we report the maximum kernel time across all participating GPU cards as the subgraph matching time. For results that run for more than an hour, we mark them with #OT.

A. Comparison with SOTA on the Triangle Counting Task

SMOG is compared against the SOTA triangle counting system (TRUST [27]), which is the extension of the champion of GraphChallenge 2019 (H-INDEX [13]), a subgraph

[1]Available at https://github.com/mengziheng/Gpu-SubgraphIsomorphism. Due to the page limit, detailed information regarding execution time and subgraph counts (including several more complex subgraphs) can also be found within the repository.

[2]Given the NP-complete nature of graph matching and identification of the automorphism group in symmetry breaking, our current focus is limited to handling small subgraphs. Nevertheless, it is interesting to tackle this challenge with medium-sized graphs and complex subgraphs, which may serve as a potential avenue for future research.



(a) cit-Patents (b) P1a

Fig. 6: The performance of SMOG with varying number of GPU cards.

matching system (RPS [30]) and a graph processing system (Gunrock [40]). Table I demonstrates the performance of these systems on the triangle counting task. Note that we omit graphs with less than 1 million edges in Table I, whose run time are always less than 1 ms. Overall, SMOG outperforms TRUST by $0.2 \times -17.94\times$ ($2.94\times$ on average), RPS by $3.35 \times -2,246.4\times$ ($203.55\times$ on average), and Gunrock by $22.87 \times -133,758.96\times$ ($35,455.52\times$ on average).

B. Comparison with RPS on Subgraph Matching Tasks

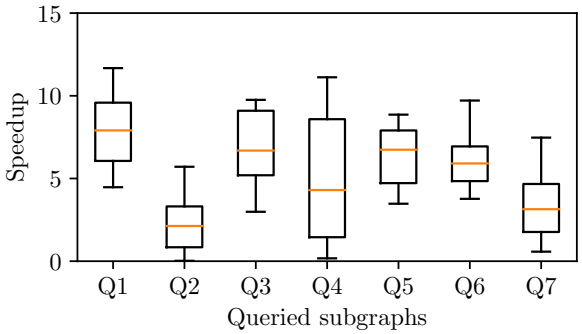


Fig. 7: Speedup ratio of SMOG over RPS.

As Gunrock performs significantly worse than SMOG, we only compare SMOG with RPS in the matching tasks of more complex subgraphs, which incur a high complexity. Figure 7 presents the speedup ratio of SMOG over RPS for various subgraphs, where each box in the boxplot depicts the distribution of speedup for Qi across real-world graphs provided by SNAP. The experimental results demonstrate that SMOG significantly outperforms RPS on cliques (Q3, Q5), thanks to the adaptive elimination strategy reducing half edges. SMOG also outperforms RPS in he remaining subgraphs due to our efficient implementation.

C. Comparison with Other GraphChallenge Systems

Due to the unavailable code or the system running on the other hardware, we compare SMOG with the results reported in the papers of these systems. FAST [21] is a subgraph matching system on CPUs that won the 2022 GraphChallenge innovation award. In roadNet-PA graph, FAST requires more than $1s$ to count the triangles while SMOG only needs $1ms$, resulting in a speedup of more than $1,000\times$. HTC [14] is a triangle counting system on GPUs that also won the 2022 GraphChallenge innovation award. HTC achieves a $1.5\times$ speedup over TRUST, while SMOG achieves $2.94\times$. Moreover, HTC can not scale to multiple GPU cards. TriC [16], [17] is a distributed triangle counting system, which won the 2020 GraphChallenge champion and 2022 innovation award. With 24 nodes and 768 processors, TriC counts triangles on the friendster graph in $16.99s$, while SMOG with 1 GPU only needs $4.77s$, resulting in a $3.6\times$ speedup.

D. Scalability

We evaluate the scalability of SMOG from 1 GPU card to 1,024 GPU cards through workload partitioning. Our strategy of workload partitioning assumes that the complete graph can fit in the GPU memory so that we can directly duplicate the entire graph across all the GPUs. Figure 6 shows the scalability of SMOG on two representative graphs from real-world and synthetic with various queried subgraphs. Particularly, SMOG achieves up to a $423\times$ speedup on cit-Patient and up to $553\times$ speedup on P1a. Several large subgraphs in cit-Patient suffer from an imbalance workload and result in poor scalability.

VI. Conclusion

In this paper, we propose SMOG, a powerful system for accelerating subgraph matching on multi-card GPUs. SMOG addresses the issue of duplication resulting from subgraph automorphism by employing a two-step approach. It begins by analyzing the symmetry within the subgraph and then adaptively adjusts the graph preprocessing and GPU code generation. By leveraging multi-level parallelism, SMOG is able to scale to 1,024 GPU cards and provides up to $553\times$ speedup from 1 to 1,024 GPU cards. Moreover, SMOG beats the triangle counting system TRUST and significantly outperforms the subgraph matching system RPS and the graph processing system Gunrock.

Acknowledgment

This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228 and 62322205.

REFERENCES

[1] A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey, "High quality, scalable and parallel community detection for large real graphs," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 225–236.

[2] A. Prat-Pérez, D. Dominguez-Sal, J.-M. Brunat, and J.-L. Larriba-Pey, "Put three and three together: Triangle-driven community detection," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 10, no. 3, pp. 1–42, 2016.

[3] H. Zhang, Y. Zhu, L. Qin, H. Cheng, and J. X. Yu, "Efficient Triangle Listing for Billion-Scale Graphs," in *Big Data*. IEEE, 2016, pp. 813–822.

[4] M. Rezvani, W. Liang, C. Liu, and J. X. Yu, "Efficient Detection of Overlapping Communities using Asymmetric Triangle Cuts," *TKDE*, vol. 30, no. 11, pp. 2093–2105, 2018.

[5] L. Zou, L. Chen, and M. T. Özsu, "Distance-join: Pattern match query in a large graph database," vol. 2, no. 1, p. 886–897, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687727

[6] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel, "SAGA: a subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 11 2006. [Online]. Available: https://doi.org/10.1093/bioinformatics/btl571

[7] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis, "Frequent substructure-based approaches for classifying chemical compounds," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 8, p. 1036–1050, 2005.

[8] B. Gaüzère, L. Brun, and D. Villemin, "Graph kernels in chemoinformatics," in *Quantitative Graph TheoryMathematical Foundations and Applications*, M. Dehmer and F. Emmert-Streib, Eds. CRC Press, 2015, pp. 425–470. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01201933

[9] J. You, J. M. Gomes-Selman, R. Ying, and J. Leskovec, "Identity-aware graph neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, 2021, pp. 10 737–10 745.

[10] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[11] Z. Chen, L. Chen, S. Villar, and J. Bruna, "Can graph neural networks count substructures?" *Advances in neural information processing systems*, vol. 33, pp. 10 383–10 395, 2020.

[12] "Graph Challenge," https://graphchallenge.mit.edu.

[13] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu, "H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs," in *HPEC*. IEEE, 2019, pp. 1–7.

[14] L. Zeng, K. Yang, H. Cai, J. Zhou, R. Zhao, and X. Chen, "Htc: Hybrid vertex-parallel and edge-parallel triangle counting," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[15] Y. Hu, P. Kumar, G. Swope, and H. H. Huang, "Trix: Triangle Counting at Extreme Scale," in *HPEC*. IEEE, 2017, pp. 1–7.

[16] S. Ghosh, "Improved distributed-memory triangle counting by exploiting the graph structure," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–6.

[17] S. Ghosh and M. Halappanavar, "Tric: Distributed-memory triangle counting by exploiting the graph structure," in *2020 IEEE high performance extreme computing conference (HPEC)*. IEEE, 2020, pp. 1–6.

[18] Y. Hu, H. Liu, and H. H. Huang, "High-performance Triangle Counting on GPUs," in *HPEC*. IEEE, 2018, pp. 1–5.

[19] T. Steil, G. Sanders, and R. Pearce, "Towards distributed square counting in large graphs," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[20] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, 2023, pp. 143–152.

[21] J. He, Z. Liu, Y. Chen, H. Pan, Z. Huang, and D. Li, "Fast: A scalable subgraph matching framework over large graphs," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[22] J. Shun and K. Tangwongsan, "Multicore Triangle Computations Without Tuning," in *ICDE*. IEEE, 2015, pp. 149–160.

[23] R. Pearce, "Triangle Counting for Scale-Free Graphs at Scale in Distributed Memory," in *HPEC*. IEEE, 2017, pp. 1–4.

[24] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: A high-performance subgraph matching system," *ACM SIGOPS Operating Systems Review*, vol. 55, pp. 21–37, 06 2021.

[25] 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.

[26] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel Triangle Counting on GPUs," in *SC*. IEEE, 2018, pp. 171–182.

[27] 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.

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

[29] 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.

[30] W. Guo, Y. Li, and K.-L. Tan, "Exploiting reuse for gpu subgraph enumeration," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 9, pp. 4231–4244, 2020.

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

[32] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *Annual International Conference on Research in Computational Molecular Biology*. Springer, 2007, pp. 92–106.

[33] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang, "Vertex priority based butterfly counting for large-scale bipartite networks," *Proceedings of the VLDB Endowment*, vol. 12, no. 10, pp. 1139–1152, 2019.

[34] T. NVIDIA, "NVIDIA A100 GPU Architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf, 2020.

[35] J. Leskovec and R. Sosič, "Snap: A General-Purpose Network Analysis and Graph-Mining Library," *TIST*, vol. 8, no. 1, p. 1, 2016.

[36] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[37] "Genbank," https://www.ncbi.nlm.nih.gov/genbank/.

[38] M. W. Group *et al.*, "MAWI Working Group Traffic Archive," https://mawi.wide.ad.jp/mawi/, 2012, accessed: 2021, April 22 [online].

[39] X. Chen and Arvind, "Efficient and scalable graph pattern mining on gpus," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 857–877. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/chen

[40] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," *SIGPLAN Not.*, vol. 51, no. 8, Feb. 2016. [Online]. Available: https://doi.org/10.1145/3016078.2851145