BUG: Balanced DFS-Based Subgraph Matching with a ReUse Strategy on GPUs

Zicang Xu
Wangxuan Institute of Computer Science and Technology
Peking University
Beijing, China
xuzicang@stu.pku.edu.cn

Lei Zou
Wangxuan Institute of Computer Science and Technology
Peking University
Beijing, China
zoulei@pku.edu.cn

***Abstract*—Subgraph matching is a fundamental problem in graph analysis with wide-ranging applications in domains such as bioinformatics, fraud detection, social networks, and recommendation systems. Despite extensive optimization efforts on CPU platforms, the NP-hard nature of subgraph matching limits the performance, especially on large-scale datasets. Leveraging the massive parallelism and high memory bandwidth of GPUs offers significant acceleration potential. However, existing GPU-based solutions face key challenges. BFS-based approaches suffer from excessive memory consumption due to the exponential growth of partial matches, while more DFS-based methods, though memory-efficient, struggle with severe load imbalance and inefficient computation reuse.**

In this work, we propose two complementary strategies to address these issues. First, we introduce a *work-offload mechanism* that dynamically balances workload across warps using a global task queue, significantly improving resource utilization. Second, we employ a combination of *symmetry breaking* and a *reuse strategy* to reduce redundant set intersections during the enumeration process. The experiments on real-world and synthetic datasets demonstrate that our approach achieves up to a 25.59× speedup compared to prior works.

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

I. Introduction

Subgraph matching is a crucial task in graph analysis, which is widely applied in various fields such as bioinformatics [1], fraud detection [2], social networks [3], and knowledge graphs [4]. Unfortunately, the subgraph matching task has been proven to be an NP-hard problem, which is computationally expensive for large datasets. During the past decades, researchers have explored extensive optimizations for handling the subgraph matching problem on CPU platforms [5]–[9], which mainly focus on reducing redundant computation. However, the performance of these methods is constrained by the computational capacity of CPUs [10].

Fortunately, with the development of GPUs, subgraph matching can be greatly accelerated on these new processors, which offer massive parallelism and high memory bandwidth. However, fully utilizing the resources of GPUs is non-trivial.

Early subgraph matching algorithms on GPUs employed a BFS search approach [11]–[13]. These solutions typically extend matches layer by layer, computing and storing all the partial matches in parallel within one kernel function at each layer. Although BFS-based solutions can leverage massive parallelism, they often suffer from excessive memory consumption due to the exponentially increasing number of partial matches [12].

To avoid excessive memory consumption, later research proposed DFS-based approaches [14]–[16]. These methods explore partial matches in a depth-first manner using a stack, which significantly reduces intermediate memory usage. Threads within a warp follow the same searching path but process different vertices when computing candidates via set intersections. However, DFS-based solutions face two key challenges. First, DFS-based methods encounter severe load imbalance despite improved memory efficiency. The number of matches extended from different root vertices varies widely due to the skewed degree distribution. As a result, most warps become idle while waiting for a few heavily loaded warps to complete, leading to poor resource utilization and degraded performance. To address this, an effective load balancing strategy is needed. Second, DFS-based solutions on GPUs cannot directly adopt the redundant computation pruning techniques developed on CPU platforms, whereas more than 70% of intersections among concurrent threads are repetitive in some cases [17]. This is due to the lack of dynamic memory allocation within GPU kernel functions at runtime. Additionally, since GPU registers are private to each thread, executing complex control flows increases register pressure, which in turn limits the number of concurrent threads. Therefore, a lightweight and efficient strategy to reduce redundant computation is essential to improve overall performance.

To address load imbalance, we proposed a **work-offload mechanism** that dynamically moves tasks from busy warps to idle warps. Specifically, we use a global task queue that stores partial matches from overloaded warps. Busy warps periodically divide their unfinished searching tasks and offload them to the task queue. In contrast, idle warps fetch tasks from the task queue and continue enumerating matches until all the warps are idled. The main challenge is to maintain the consistency of the task queue without affecting the matching logic severely.

To reduce redundant computation, we adopt a **symmetry-breaking strategy** to avoid multiple enumeration due to the automorphism of the query graph. We further explore combining a **reuse strategy** to reduce the recomputation of

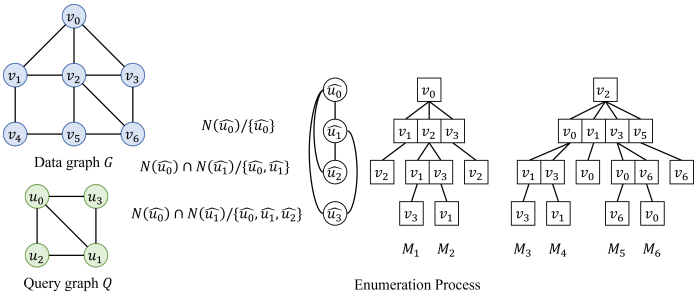


Fig. 1: An example of Subgraph Matching.

set intersections. Specifically, we can regard the searching process as a sequence of set intersection operations. The reuse mechanism attempts to detect intersection operations with the same sets and reuse the results. These strategies can reduce the amount of computation and IO cost, which are orthogonal to the work-offload mechanism.

The contribution of this paper can be summarized as follows:

- We analyze the prior subgraph matching algorithms and point out the load imbalance and redundant computation problem in the DFS-based works.
- We design a work-offload mechanism to address the load imbalance problem. And we integrate symmetry-breaking and a reuse strategy to reduce redundant computation. Based on these, we proposed BUG, which is a balanced and DFS-based subgraph matching solution with a reuse strategy on GPU.
- We conduct extensive evaluations on a wide range of both real-world and synthetic datasets. The experimental results show that BUG can achieve up to $3.40\times$ speedup compared to prior works on triangle counting tasks, and up to $25.59\times$ speedup on general subgraph matching tasks.

II. Background and Motivation

A. Preliminaries

We use $G = (V_G, E_G)$ and $Q = (V_Q, E_Q)$ to represent the data graph and the query graph, where V denotes the vertex set and $E \subseteq V \times V$ denotes the edge set of G and Q. We use $N(u)$ to denote the neighbors of vertex $u \in V$. The subgraph matching problem requires finding all the subgraph isomorphisms, which is defined below:

Definition 2.1 (Subgraph Isomorphism): Given a data graph $G = (V_G, E_G)$ and a query graph $Q = (V_Q, E_Q)$, subgraph isomorphism is an injective mapping $f : V_Q \to V_G$, such that $\forall (u_1, u_2) \in E_Q, (f(u_1), f(u_2)) \in E_G$.

For simplicity of representation, we also use $\hat{u_i}$ to denote $f(u_i)$.

When enumerating matches of the query graph, existing algorithms often follow a specific order to extend candidate vertices based on the current partial match.

Definition 2.2 (Matching order): For a given query graph Q, a matching order π is a function reflecting the order the mapping vertices of V_Q are assigned. We use $\pi(i)$ to denote the i^{th} vertex to be mapped, and $\pi^{-1}(u)$ to denote the rank of vertex u in the matching order.

Definition 2.3 (Backward Neighborhood): In the query graph, the backward neighbor of a vertex $u \in V_Q$ for a given π is the set $BN(u) = \{w | w \in N(u) \wedge \pi^{-1}(w) < \pi^{-1}(u)\}$

B. Related Work

To accelerate subgraph matching on large graphs, recent research has turned to GPU platforms, which offer massive parallelism and high memory bandwidth. There are two mainstream strategies for enumerating matches on GPUs: BFS-based and DFS-based.

Fig. 1 illustrates an example of subgraph matching. With a given π, the enumeration process can be regarded as traversing a search tree. The two strategies differ in how they traverse the trees.

BFS-based solutions, such as GPSM [10], GSI [11], and RPS [13], expand partial matches level by level and fully exploit GPU parallelism by processing the partial matches concurrently. However, due to the combinatorial explosion of partial matches at intermediate levels, BFS-based solutions failed to maintain all the partial matches in the GPU's memory. Therefore, researchers turn to explore DFS-based solutions to address the memory problem.

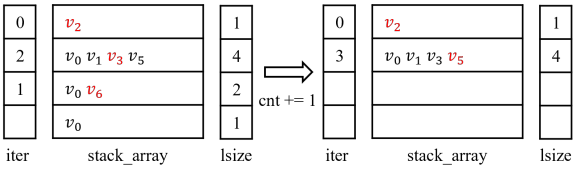


Fig. 2: A stack example for the enumeration process.

DFS-based solutions, such as PARSEC [14] and SMOG [16], adopt a warp-level task scheduling. Threads within a warp share the same traversal path when enumerating and handle different vertices during intersections. Initial matches are assigned to different warps for warp-level parallelism. To implement a DFS strategy on GPUs, they allocate a stack for each warp, which stores intermediate results for each layer and maintains the state of the traversal. Fig. 2 illustrates an example of a warp stack. The `stack_array` stores the intermediate results at each level, which is used for set operations and tree traversal. The depth of `stack_array` is determined by the size of the query graph, while its width is usually set as the maximum degree of the data graph. `lsize` and `iter` are two arrays located in the shared memory, which record the number of candidate vertices and the loop iterator for each level. The example in Fig. 2 is handling the partial match $[v_2, v_3, v_6]$. It calculates $N(v_2) \cap N(v_3) \setminus \{v_2, v_3, v_6\}$ and find a match $[v_2, v_3, v_6, v_0]$. Then it adds the global match counter and moves to the next partial match $[v_2, v_5]$.

DFS-based solutions consume $O(|V_Q| \times max\_deg)$ memory for warp stacks, which is a fixed and relatively small space for global memory. However, there are two possible optimizations

Statistic	Q7	Q8	Q9
Mean	199.48	45.87	53.27
Median	199.12	45.75	50.65
99th Percentile	199.12	47.06	123.22
Maximum	683.29	80.79	313.31

TABLE I: Execution time statistics for Q7-Q9(see Fig. 7) on the cit-Patents dataset. All values are in million cycles.

for DFS-based solutions. First, the imbalance among different enumeration trees becomes severe as $|V_Q|$ increases, causing resource underutilization. We test typical query graphs(see Fig. 7) on cit-Patents [18], and record the execution time of each warp. Table I shows some statistics of the execution time. For complex query graphs, a few warps execute much longer than other warps, which prolongs the total execution time of the kernel function. Therefore, we design a work-offload mechanism to move tasks from overloaded warps to idle warps adaptively, which is more efficient and flexible than the workload balancing technique of prior works [15], [19]. The second optimization is reducing redundant computation. SMOG has leveraged symmetry breaking to avoid enumerating automorphic matches by applying order restrictions on equivalent vertices. We further combine it with a reuse mechanism based on the inclusion relation of the backward neighborhood of query vertices. Take the query graph shown in Fig. 1 as an example, computing the candidate vertices of u_3 can reuse the intermediate results of u_2 since they both need to compute $N(\hat{u_0}) \cap N(\hat{u_1})$. The details of the two mechanisms are introduced in Section III.

III. Methodology

A. Algorithm Overview

Fig. 3 illustrates the framework of BUG. The whole framework comprises two stages: preprocessing and enumeration. In the preprocessing stage, we analyze the input query graph to generate a matching plan. Specifically, we first generate a matching order as prior works have adopted [11], which sequentially selects the next vertex with the largest $|BN(u)|$ and $|N(u)|$. Then, we compute automorphisms on the query graph and add order restrictions to implement symmetry breaking. Finally, we detect the reusable intersect operations based on the backward neighborhood. In the enumeration stage, warps adaptively fetch tasks(initial matches) and enumerate matches extended from them. If a warp works on a task for a long time, it proactively divides its remaining work into subtasks and offloads them to a task queue. When a warp completes its task, it consistently attempts to fetch tasks from the task queue until all the warps are idle and the task queue is empty. The enumeration process is similar to prior DFS-based work, except for a reuse mechanism when computing candidates at certain levels.

B. Work-Offload Mechanism

We maintain a timer for each warp to record the execution time of a task. CUDA provides a time function `clock()` that

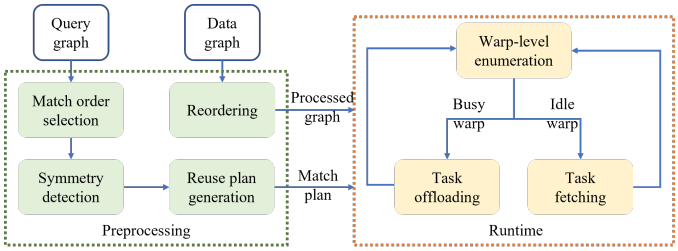


Fig. 3: BUG framework.

reports the current clock cycle of the multiprocessor. When a warp starts working on a task, it sets its timer by using the `clock()` function. Each time the warp starts extending a new vertex, it obtains the current clock cycle and checks whether the execution time of the current task exceeds a threshold(e.g., 1 million cycles). If so, it finds the highest level having unprocessed vertices and offloads the partial matches at that level to a task queue. Fig. 4 shows an example of the offload procedure. When the warp detects a timeout, it checks the `iter` and `lsize` array and finds that the second layer has three unprocessed partial matches. The partial matches are regarded as three tasks and are offloaded to the task queue. Then the warp resets its timer and the second element of `lsize`, and continues its enumeration process.

The task queue contains an array of task slots, which is located in the global memory. We use two pointers `head` and `tail` to manage which slot should be read or written. To avoid blocking in the queue, we use `task_count` to prevent reader or writer warp from accessing when the task queue is empty or full. Each task slot in the queue contains a vertex array to store the partial match and a `size` field used for indicating the size of the partial match as well as the status of the slot.

Algorithm 1 shows two basic operations, `dequeue` and `enqueue`, supported by the task queue. Both operations are executed by a single warp. The `enqueue` operation adds multiple tasks at one time, while the `dequeue` operation consumes a single task from the queue. For simplicity, the atomic operations in Algorithm 1 are directly written while they are only executed by the leader thread in practice.

When a warp executes the `enqueue` operation, it first reserves enough slots for work offloading(line 2). If the queue has insufficient capacity to accommodate the tasks, the reservation is rolled back, and the operation fails. The warp then resets its timer, double its time threshold, and continues to enumerate(lines 3-5). Otherwise, the warp atomically acquires a contiguous range of slots by updating the tail pointer(line 6). For each slot, the warp uses `atomicCAS` to ensure exclusive write access(line 10). Once a slot is successfully acquired, the warp writes the task data into it and marks it as readable by setting the `size` field(lines 11-12). Since GPUs support fast atomic operations, the warp can only be blocked at lines 9-10. However, since we constrained the number of warps accessing the queue, only one warp may be reading the slot, which will soon be done. Therefore, the `enqueue` operation

is nearly lock-free and can be ignored if we use a relatively high threshold for the timer.

The `dequeue` operation similarly checks whether there are tasks to read(lines 15-18). It then obtains a slot from the `head` pointer and waits until the slot becomes valid(lines 19-21). Then it reads the task in the slot, and sets the `size` field to 0, indicating that the slot is consumed and can be rewritten.

Our work-offload mechanism design has three advantages: first, the offload tasks can be partial matches of any length, which is generic and flexible. Second, the remaining work was equally divided at the highest level and was offloaded to the task queue at one time, achieving a good load-balancing effect. Third, the `enqueue` operation is nearly lock-free and does not heavily affect the enumeration performance.

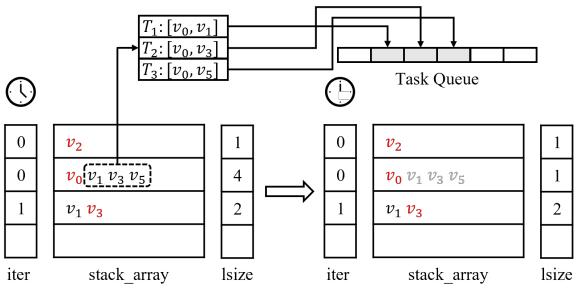


Fig. 4: An example of work offload procedure.

C. Symmetry-breaking and Reuse Strategy

To reduce the amount of computation, prior works have proposed symmetry-breaking and reuse strategies [15], [17], which we also adopt in this paper.

The symmetry-breaking strategy selects a unique representative element from the automorphism group of the query graph. For example, consider the query graph in Fig. 1. The symmetry-breaking strategy imposes order constraints on equivalent vertices, such as $\hat{u_1} < \hat{u_0}$ and $\hat{u_3} < \hat{u_2}$. Under these restrictions, the first enumeration tree is pruned at the second level. This strategy can be easily integrated into DFS-based subgraph matching on GPUs by simply discarding vertices that violate the imposed constraints during candidate computation.

The reuse strategy leverages the inclusion relationship between the vertices of the query graph. Again taking Fig. 1 as an example, the candidates of u_2 are computed as $N(\hat{u_0}) \cap N(\hat{u_1}) \setminus \{\hat{u_0}, \hat{u_1}\}$, while those of u_3 are computed as $N(\hat{u_0}) \cap N(\hat{u_1}) \setminus \{\hat{u_0}, \hat{u_1}, \hat{u_2}\}$. Therefore, we can use $C(u_2) \setminus \{\hat{u_2}\}$ to compute the candidates of u_3, where $C(u_2)$ denotes the current candidate set of u_2. In general, if $BN(u_i) \subseteq BN(u_j)$ for $i < j$, the candidate set of u_i can be reused to compute that of u_j.

To implement the reuse strategy, we generate a reuse plan in the preprocessing stage. For each $u_j \in V_Q$, we select the best reusable vertex. Specifically, we find all the $u_i (i < j)$, such that $BN(u_i) \subseteq BN(u_j)$ and select the one with the largest $|BN(u_i)|$ as the u_j's reuse vertex. During the enumeration stage, the candidate set of u_j can be computed as

$$C(u_i) \cap (\cap_{k \in BN(u_j) \setminus BN(u_i)} N(u_k)) \setminus \{u_h | i \leq h < j\}.$$

Algorithm 1: Warp-level `enqueue` and `dequeue` operations for the task queue

```
1  Function enqueue(T[], l, num):
2      fill ← atomicAdd(task_count, num);
3      if fill + num > N then
4          atomicSub(task_count, num);
5          return false;
6      start_pos ← atomicAdd(tail, num)%N;
7      for i ← 0 to num − 1 do
8          p ← (start_pos + i)%N;
9          while atomicCAS(A[p].size, 0, −1) ≠ 0 do
10             __nanosleep(10);
11         write task T[i] into A[p].array;
12         T[i].size ← l;
13     return true;
14 Function dequeue(&T, &l):
15     readable ← atomicSub(task_count, 1);
16     if readable ≤ 0 then
17         atomicAdd(task_count, 1);
18         return false;
19     pos ← atomicAdd(head, 1)%N;
20     while A[pos].size == −1 or 0 do
21         __nanosleep(10);
22     T ← A[pos].array, l ← A[pos].size;
23     A[pos].size ← 0
24     return true;
```

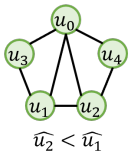


Fig. 5: An example query graph for symmetry-breaking and reuse strategy.

And the $C(u_i)$ is the candidate set of u_i and was stored in the i^{th} level of the warp stack.

We integrate the reuse strategy with both the symmetry-breaking and work-offload mechanisms. Two aspects need to be considered: first, the reuse vertex may have different restrictions, which may make the candidate set not directly reusable. Second, when a task is transferred to another warp through the work-offload mechanism, the candidate sets stored in the stack are changed and may not be able to be reused.

We first demonstrate how to address the first problem. Suppose u_i is reused by $u_{j_1}, ..., u_{j_k}$ in the reuse plan, where $j_1, ..., j_k > i$. If all these vertices have an order relation with u_i in the symmetry restriction, the candidate set of u_i under the restriction can be directly reused. Otherwise, we compute and store all the candidates of u_i, but only traverse the ones satisfying the restriction. Take the query graph shown in Fig. 5

TABLE II: GPU runtime of BUG, SMOG, TRUST, RPS on triangle counting task (in ms).

Datasets	#Vertices	#Edges	#Triangles	BUG	SMOG		TRUST		RPS	
				Time (ms)	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup
SNAP Datasets										
amazon0302	262,111	899,792	717,719	3.39	1.22	0.36	0.76	0.22	17.69	5.22
amazon0505	410,236	2,439,437	3,951,063	3.43	3.33	0.97	1.33	0.38	22.55	6.57
amazon0601	403,394	2,443,408	3,986,507	3.43	3.33	0.97	1.23	0.36	19.83	5.78
cit-HepPh	34,546	420,877	1,276,868	4.48	0.77	0.17	0.62	0.13	14.85	3.31
cit-HepTh	27,769	352,285	1,478,735	3.47	0.69	0.29	0.61	0.17	14.32	4.12
cit-Patents	3,774,768	16,518,947	7,515,023	7.12	22.26	3.12	2.15	0.30	153.47	21.55
flickrEdges	105,718	2,081,520	107,987	11.41	6.90	0.60	4.64	0.41	33.33	2.92
roadNet-CA	1,965,206	2,766,607	120,676	3.53	2.83	0.80	1.70	0.48	26.93	7.62
roadNet-PA	1,088,092	1,541,898	67,150	3.56	1.66	0.46	0.96	0.27	21.05	5.91
email-Enron	36,692	183,881	727,044	3.69	0.48	0.14	0.46	0.12	13.45	3.64
Synthetic Kronecker Datasets										
25-81-256-B1k	547,924	2,132,284	2,102,761	5.96	2.25	0.38	1.44	0.24	43.13	7.23
25-81-256-B2k	547,924	2,132,284	7	4.57	1.23	0.27	0.57	0.12	43.28	9.47
3-4-5-9-16-25-B1k	530,400	11,080,030	35,882,427	24.11	27.27	1.13	16.61	0.69	165.92	6.89
3-4-5-9-16-25-B2k	530,400	11,080,030	651	13.38	11.78	0.88	6.33	0.47	97.10	7.26
4-5-9-16-25-B1k	132,600	1,582,861	3,548,463	5.25	2.50	0.48	2.06	0.39	27.12	5.17
4-5-9-16-25-B2k	132,600	1,582,861	155	4.68	1.33	0.28	0.93	0.20	22.57	4.82
5-9-16-25-81-B1k	2,174,640	28,667,380	66,758,995	115.76	63.56	0.55	43.09	0.37	609.93	5.27
5-9-16-25-81-B2k	2,174,640	28,667,380	155	62.77	26.82	0.43	14.00	0.22	450.63	7.18
9-16-25-81-B1k	362,440	2,606,125	4,059,175	7.25	3.63	0.50	2.18	0.30	45.40	6.26
9-16-25-81-B2k	362,440	2,606,125	35	6.27	1.87	0.30	0.95	0.15	39.38	6.28
MAWI Datasets										
201512012345	18,571,154	19,020,160	2	8.48	1.35	0.16	0.94	0.11	821.16	96.83
201512020000	35,991,342	37,242,710	2	12.24	1.61	0.13	1.31	0.11	1563.67	127.83
201512020030	68,863,315	71,707,480	6	19.79	2.91	0.15	1.67	0.08	2913.33	147.21
201512020130	82,389,971	89,009,590	10	34.11	9.69	0.28	3.18	0.09	5416.28	158.79
201512020330	226,196,184	240,023,945	26	58.84	11.78	0.20	5.73	0.10	9546.23	162.24
GenBank Datasets										
P1a	139,353,211	148,914,992	3,412	39.20	147.29	3.76	96.65	2.46	1315.47	33.55
U1a	67,716,231	69,389,281	325	20.61	67.56	3.27	29.82	1.44	564.95	27.41
V1r	214,005,017	232,705,452	49	59.37	221.12	3.72	145.28	2.45	1779.87	29.96
V2a	55,042,369	58,608,800	1,443	14.94	48.58	3.25	21.47	1.44	456.72	30.57
graph500 Datasets										
scale18-ef16	174,147	3,800,348	82,287,285	8.49	7.62	0.90	10.27	1.21	52.75	6.21
scale19-ef16	335,318	7,729,675	186,288,972	14.43	19.72	1.37	25.16	1.74	109.78	7.61
scale20-ef16	645,820	15,680,861	419,349,784	26.98	52.38	1.94	65.26	2.42	331.10	12.27
scale21-ef16	1,243,072	31,731,650	935,100,883	80.40	199.68	2.48	133.22	1.66	685.65	8.53
scale22-ef16	2,393,285	64,097,004	2,067,392,370	185.35	423.21	2.28	263.26	1.42	1535.98	8.29
scale23-ef16	4,606,314	129,250,705	4,549,133,002	420.36	1138.62	2.71	676.97	1.61	3696.54	8.79
scale24-ef16	8,860,450	260,261,843	9,936,161,560	848.62	2883.13	3.40	1766.60	2.08	8726.74	10.28

as an example, we find that u_3 can leverage the intermediate result of u_2, for they both calculate $N(u_0) \cap N(u_1) \setminus \{u_0, u_1\}$. However, we apply a restriction $\hat{u_2} < \hat{u_1}$ when mapping u_2. Without the reuse strategy, we directly prune candidates that violate the constraints. But when applying the reuse strategy, we store all the vertices in the stack, but the corresponding element of `lsize` is still set as the number of candidates satisfying the restriction. Moreover, we use another array `lsize_origin` to indicate the number of total candidates.

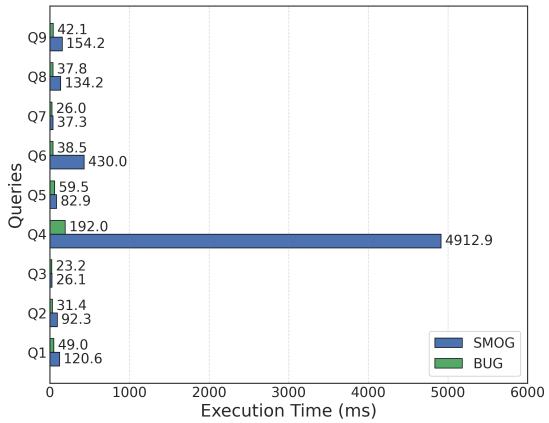
To address the second issue, we discuss the offloading and fetching of warps separately. As we have added a new array `lsize_origin`, which is not changed after task offloading, the stack of the offloading warp is still reusable. For the fetching warp, we switch the candidate computation mode if the reuse vertex is not in the current stack. We also use the query graph shown in Fig. 5 as an example. If a warp fetches a task of a 2-vertex partial match $[\hat{u_0}, \hat{u_1}]$, it can still reuse the candidate of $\hat{u_2}$ when processing u_3. If the task is a 3-node partial match $[\hat{u_0}, \hat{u_1}, \hat{u_2}]$, the algorithm switches to compute $\hat{u_2}$ without applying the reuse strategy.

IV. Evaluation

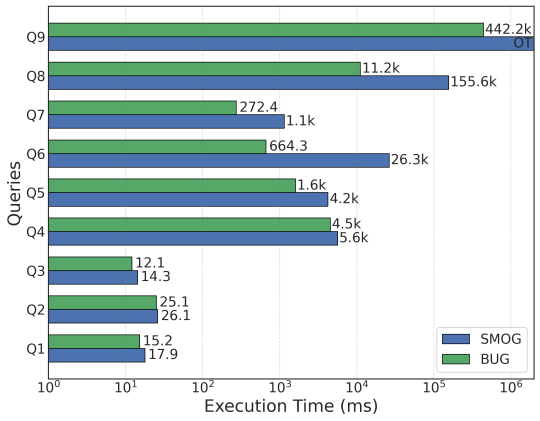
In this section, we show the efficiency of BUG with extensive experiments. We use the datasets provided by the GraphChallenge, including SNAP datasets [18], synthetic Kronecker graphs [20], protein k-mer graphs [21], and MAWI graphs [22]. We test the performance of triangle counting and generic subgraph matching tasks as shown in Fig. 7. The experiments are conducted on an A100 GPU, equipped with 40GB of device memory. We report the execution time of the GPU kernel functions following prior works of GraphChallenge.

A. Experiments on the Triangle Counting Task

We compare BUG with prior triangle counting and subgraph matching systems, including SMOG [16], TRUST [23],



(a) Execution time on the cit-Patent dataset.



(b) Execution time on the ca-HepPh dataset.

Fig. 6: Comparison of execution times

and RPS [13]. Table II illustrates the performance of these systems on the triangle counting task. The results show that BUG achieves up to $3.76\times$ speedup compared to SMOG, $2.46\times$ compared to TRUST, and $162.24\times$ compared to RPS. Although our approach performs worse on some datasets, it performs well on large graphs with massive triangles such as GenBank datasets and graph500 datasets. On smaller graphs, performance degradation may occur due to complex control flow. In contrast, larger graphs yield more balanced workloads for our algorithms. Furthermore, BUG leverages binary search for set intersections, which can restrict the search space of the vertex index in triangle counting and reduce computational overhead.

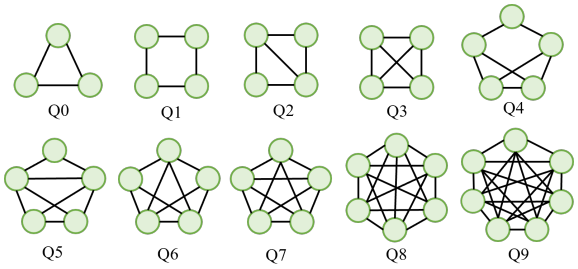


Fig. 7: Query graphs for generic subgraph matching.

B. Experiments on General Subgraph Matching Tasks

We compare BUG with SMOG on general subgraph matching tasks using the query graphs shown in Fig. 7. We select cit-Patent and ca-HepPh datasets as the data graphs, which are real-world graphs provided by SNAP. Fig.6 shows the performance of BUG and SMOG of querying Q1-Q9. The results show that BUG outperforms SMOG by $1.13\times - 25.59\times$ on the cit-Patent dataset, and $1.03\times - 13.89\times$ on the ca-HepPh dataset. This is because the skewness increases significantly as the size of query graphs increases, and SMOG fails to balance the workload among warps.

C. Ablation Study

We conduct an ablation study on the two strategies we proposed in this work. Since symmetry-breaking is crucial for completing queries in an acceptable time, it is applied to all the settings we test. We handle Q5-Q9 on the ca-HepPh dataset. The results are shown in Fig. 8. It can be seen that, on Q6-Q7, both the work-offload mechanism and the reuse strategy improve the performance. However, on Q5, the work-offload mechanism causes a performance degradation. This is because the work-offload mechanism increases the register requirement of each thread, limiting the number of threads within a block.

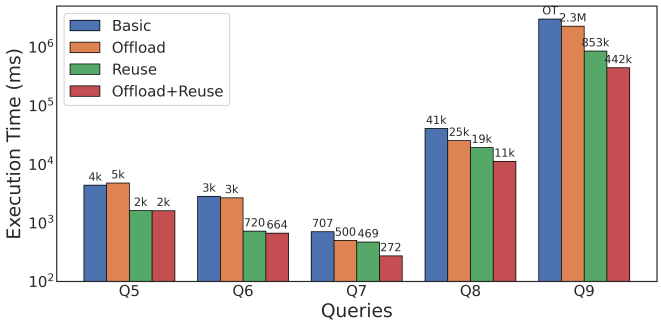


Fig. 8: Query graphs for generic subgraph matching.

V. Conclusion

In this paper, we propose BUG, an efficient subgraph matching system on GPUs. BUG employs a work-offload mechanism to dynamically redistribute tasks from overloaded warps to idle ones, effectively mitigating load imbalance and improving overall warp utilization. In addition, we combine the symmetry breaking and a reuse strategy to reduce redundant set intersection operations, enhancing computational efficiency without introducing significant overhead. Extensive experiments demonstrate that our solution achieves superior performance compared to state-of-the-art methods, including SMOG, TRUST, and RPS.

REFERENCES

[1] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC Bioinformatics*, vol. 14, pp. S13 – S13, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:1338930

[2] A. Quamar, A. Deshpande, and J. Lin, "Nscale: Neighborhood-centric large-scale graph analytics in the cloud," 2015. [Online]. Available: https://arxiv.org/abs/1405.1499

[3] W. Fan, "Graph pattern matching revised for social network analysis," in *Proceedings of the 15th International Conference on Database Theory*, ser. ICDT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 8–21. [Online]. Available: https://doi.org/10.1145/2274576.2274578

[4] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, "Taming subgraph isomorphism for rdf query processing," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1238–1249, Jul. 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809985

[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] 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.

[7] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 804–818, 2017.

[8] 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.

[9] A. Jüttner and P. Madarasi, "Vf2++—an improved subgraph isomorphism algorithm," *Discrete Applied Mathematics*, vol. 242, pp. 69–81, 2018.

[10] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Database Systems for Advanced Applications*, M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, Eds. Cham: Springer International Publishing, 2015, pp. 299–315.

[11] 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)*, 2020, pp. 1249–1260.

[12] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam, "cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476214

[13] 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, 2022.

[14] V. Dodeja, M. Almasri, R. Nagi, J. Xiong, and W.-m. Hwu, "Parsec: Parallel subgraph enumeration in cuda," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 168–178.

[15] Y. Wei and P. Jiang, "Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[16] Z. Wang, Z. Meng, X. Li, X. Lin, L. Zheng, C. Tian, and S. Zhong, "Smog: Accelerating subgraph matching on gpus," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–7.

[17] S. Kawtikwar, M. Almasri, W.-M. Hwu, R. Nagi, and J. Xiong, "Beep: Balanced efficient subgraph enumeration in parallel," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 142–152. [Online]. Available: https://doi.org/10.1145/3605573.3605653

[18] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2898361

[19] L. Yuan, D. Yan, J. Han, A. Ahmad, Y. Zhou, and Z. Jiang, "Faster depth-first subgraph matching on gpus," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 3151–3163.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," 2009. [Online]. Available: https://arxiv.org/abs/0812.4905

[21] Genbank. National Center for Biotechnology Information. [Online]. Available: https://www.ncbi.nlm.nih.gov/genbank/

[22] Mawi. National Center for Biotechnology Information. [Online]. Available: https://mawi.wide.ad.jp/mawi/

[23] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, and H. Liu, " Trust: Triangle Counting Reloaded on GPUs ," *IEEE Transactions on Parallel & Distributed Systems*, vol. 32, no. 11, pp. 2646–2660, Nov. 2021. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3064892