# MERCURY: Efficient Subgraph Matching on GPUs with Hybrid Scheduling

Zhiheng Lin, Changjie Xu, Ke Meng, Guangming Tan
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences
lingzhiheng@ncic.ac.cn, xuchangjie21@mails.ucas.ac.cn, {mengke, tgm}@ict.ac.cn

***Abstract*—Subgraph matching finds all distinct subgraphs in the given data graph $G$ that are isomorphic to the pattern graph $P$. It is widely used in social networks, chemoinformatics, recommendation systems, anomaly detection, and network security. Unfortunately, subgraph matching is an NP problem with a huge search space that can quickly exhaust computational resources and requires materializing a large number of intermediate results. Even with GPU acceleration, the processing time for subgraph matching tasks on large graphs often fails to meet the needs of real-world applications. Previous systems use coarse-grained thread-mapping strategies and static configuration for symmetry breaking rules and intersection kernels, which lose the opportunity to exploit the fine-grained parallelism of GPUs. In this paper, we first discuss different optimization variants from three aspects:(i) symmetry breaking, (ii) thread-mapping and (iii) intersection kernel, then we propose a novel hybrid scheduling strategy to combine these optimization variants. Based on this scheduling, we developed MERCURY to enable load-balanced and efficient subgraph matching on GPUs. Experiments show that MERCURY outperforms TRUST, SMOG, and H-INDEX up to 3.92×, 19.7×, and 21.6×, respectively, in triangle counting. For general pattern matching tasks, it is up to 52.4× faster than G$^2$Miner, and can scale up to 1024 GPU cards.**

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

## I. INTRODUCTION

Graphs, as a flexible data structure, that can efficiently represent the relationships between entities, have been widely adopted to model real-world data. Subgraph matching tasks aim to find all subgraphs in a given data graph $G$ that are isomorphic to a pattern graph $P$, where the size of $G$ is significantly larger than $P$. Such subgraph matching tasks can uncover hidden data characteristics or find specific matches and are therefore widely used in various data analysis tasks, such as community detection [1], [2], molecular dynamics [3], drug discovery [4], and anomaly detection [5], [6].

The subgraph matching problem has been studied for decades, but no polynomial-time solution has been found yet. Practically, the subgraph matching problem is a search problem that starts from every vertex in the graph, iteratively adding adjacent edges, and checking whether the newly formed candidate subgraph is isomorphic to the target pattern $P$. This method not only requires traversing a huge search space but also materializing a large number of intermediate results. GPUs, with their numerous cores capable of simultaneous enumeration and TB-level bandwidth provided by High Bandwidth Memory (HBM) [7], are potential devices to accelerate the subgraph matching problem from a performance perspective.

Recently, many GPU-based subgraph matching systems have been developed to accelerate this type of application. The core of these systems centers on three aspects: (i) How to avoid automorphism, which causes redundant computation. GraphPi [8] reduces automorphisms by adding symmetry-breaking rules, TRUST [9] uses a vertex ordering strategy called orientation pruning to preemptively shrink the neighbor lists thus avoiding touching automorphism triangles. (ii) How to expedite the set intersection kernel that is invoked millions of times during the search process to determine whether two points in a graph share any common neighbors. For instance, TriCore [10] employs hash-based intersection, MergePath [11] uses merge-based intersection, Gunrock [12] utilizes binary search-based intersection, and GraphFold [13] adopts a reverse intersection approach. (iii) How to efficiently map the search task to the GPU threads, enabling the Graph Pattern Matching (GPM) algorithms to reach the hardware limits of GPUs as far as possible. For example, G$^2$Miner [14] generates GPU-friendly code via code generation, GraphFold [13] avoids redundant edge checking in intersections. Due to the power-law edge distribution in the real-world graph, the computational resources required to compute the intersection of neighbors for two vertices can vary significantly—some intersections may involve traversing thousands of vertex pairs, while others may require only a few. Such highly skewed edge distribution makes one can not simply choose one optimization to rule all input graphs and patterns.

However, previous systems, including the champions of HPEC Graph Challenges, have only focused on a single optimization or coarse-grained hybrid optimizations in these three aspects mentioned above (*i.e.,* orientation pruning, intersection kernel, and thread mapping). They have not considered the performance behavior of different optimizations on different input graphs and patterns. For example, the classical thread-mapping strategy in these systems is dividing the computational resources of GPU into three categories: thread, warp, and block. They then classify based on the vertex degree, *i.e.,* using a block to handle a high-degree vertex, a thread to handle a low-degree vertex, and the remaining vertices are handled by warps individually. Since the computation of the intersection kernel often happens between vertex sets, *e.g.,* $N(v)$ and $N(u)$, Such strategy maps tasks only based on

$|N(v)|$, but still ignores the skewed distribution of $|N(u)|$.

**MERCURY**. To further improve the efficiency of GPU-based subgraph matching, we developed MERCURY[1]. MERCURY adopts a hybrid scheduling approach, which changes the static setting of optimizations in orientation pruning, intersection kernel, and thread mapping to hybrid versions, This approach makes MERCURY more robust to different input graphs and patterns thus we can achieve higher GPU utilization and performance in large GPU clusters. To be more specific, the contributions of this paper are:

(1) We conduct a detailed analysis of subgraph matching algorithm optimizations from three aspects: orientation pruning, intersection kernel, and thread mapping, and describe their performance behavior and explain why previous systems apply them. (Section II).

(2) We add new optimization variants to existing optimizations and propose a hybrid scheduling strategy, achieving higher GPU Utilization and avoiding load imbalance. (Section III).

(3) Experimental results show that MERCURY can significantly outperform the performance of SOTA and HPEC Graph Challenge champions (Section IV).

## II. Background and Motivation

In this section, we first introduce the characteristics of subgraph matching applications and the crucial intersection kernel. Then we analyze the coarse-grained load-balance algorithm and show why it fails to achieve optimal performance.

### A. Set-centric Graph Pattern Matching

A graph $G = (V, E)$ donates a set of vertices $V$ and edges $E \subseteq V \times V$, and real-world graph is often very sparse, which means $|E| \ll |V| \times |V|$. In addition, $v \sim u$ means $v$ is connected to $u$ and $v \nsim u$ is the opposite. We use $N(v)$ to represent all the neighbors of $v$. *w.l.o.g.*, we only consider unlabeled and undirected patterns and graphs, and all data can fit the aggregated device memory.

A subgraph matching algorithm is to find or count the number of all distinct embeddings, where an embedding $E_{P_k}$ is an instance of the subgraph pattern that is isomorphic with the given pattern $P_k$ on $k$ ($k > 2$) vertices, *e.g.,* clique-counting and motif-counting. A typical subgraph matching algorithm follows a 'generate-check' procedure, it first generates a candidate set $S$ and then checks all candidates in $S$ whether it can form a valid embedding. Recent studies use a set-centric model to represent the subgraph matching algorithm. As shown in Fig. 1 (a), given a $k$-vertices pattern $P_k$ with a matching order $Z$ and a data graph $G$, $E_{P_i}$ is an embedding of $P_k$. A function $\mathcal{F}^i(E_{P_i})$ is introduced to generate the candidate set $S(v_i)$, and we can use only set intersection and set difference to implement $\mathcal{F}^i(E_{P_i})$ [15]. We also follow this paradigm in this paper.

Checking the common-neighbor of a given vertex pair are essential kernel in subgraph matching algorithms. Since the pattern graph is undirected, there are many automorphisms in search space, which means the same embedding can be generated in different ways. To break the automorphism, we can put a restriction $id(v_i) < id(v_j)$, where $id(v_i)$ is the unique identifier of vertex $v_i$ in the pattern graph. This symmetric breaking optimization, which is also called orientation optimization [16], [9], is able to prune the search space but is orthogonal to MERCURY.

### B. Intersection Kernel

The operation of set intersection is a key kernel in subgraph matching, primarily occurring in the computation of the set operations described in Fig. 1 (a). The input to the intersection operation consists of two sorted arrays of positive integers, A and B, and the algorithm outputs the common elements of these two arrays. In a multicore architecture, this is typically achieved through the following two approaches:

**Binary Search**. Since arrays A and B are sorted, we can enumerate each value in the shorter of the two arrays and perform a binary search in the longer array. *w.l.o.g.*, let's assume the length of array A is less than that of B, the time complexity for each thread during the search is $O(log(len(B)))$. Due to the extensive random memory accesses required by binary search, the actual memory bandwidth is significantly lower than the theoretical peak. Therefore, in practice, part of array B is often cached in shared memory to improve performance. Specifically, the upper layers of the binary search tree constructed from array B are cached. Systems such as G$^2$Miner [14], GraphFold [13], AutoMine [15], GraphPi [8], and Gunrock [12] employ this method.

**Hash Table**. Hash-based set intersection implementation requires constructing a hash table for one of the arrays, either A or B, at runtime. Specifically, open addressing is typically used to handle hash collisions due to its straightforward implementation, which is favorable for multicore architectures. When a collision occurs, linear probing is employed to store conflicting values. In GPU implementations, managing linked list structures incurs high performance overhead, so multiple slots are usually reserved for each hash value to handle collisions, leading to significant memory consumption. Similarly, to further reduce memory access overhead during queries, part of the hash table can be stored in the shared memory of GPU. Systems such as TRUST [9], GraphFold [13], pbitMCE [17], TriCore [10], and SMOG [16] employ this method.

### C. Coarse-Grained Load-Balance Strategies

In a GPU, the relatively weak computational power of a single core means that an imbalanced workload can lead to significant wastage of computational resources, thereby greatly hurting the end-to-end performance. Therefore, determining how to distribute the workload among each thread and how to organize threads into a reasonable granularity to access and process data plays a crucial role in achieving the peak performance of the hardware.

**Edge-parallel and Vertex-parallel**. In vertex-parallel parallelism, task allocation involves assigning the processing of

[1]Code available in `https://github.com/GPM-lib/Mercury`

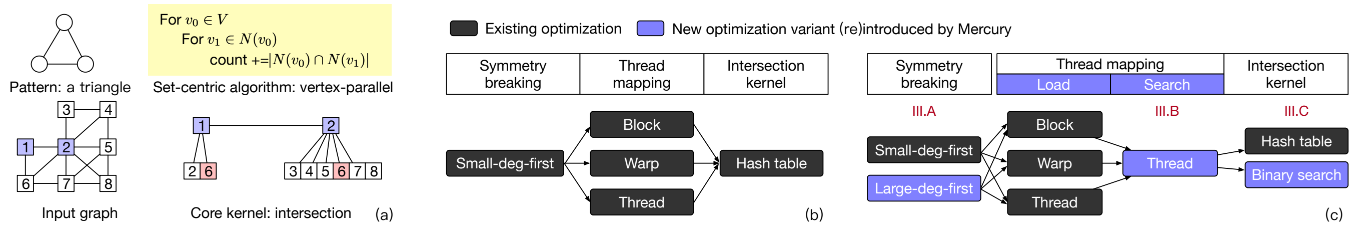

Fig. 1: (a) Subgraph matching algorithm and intersection. (b) Optimizations in previous work (c) New optimization variants (re)introduced in this paper.

a single vertex as the smallest task granularity to a thread group, whereas in edge-centric parallelism, the smallest task granularity is an adjacent edge of that vertex, which is then assigned to a thread group. Using vertex-parallel parallelism can lead to stragglers due to varying degrees of vertices. On the other hand, edge-parallel parallelism, with its finer granularity, can achieve better load balancing. However, edge-parallel parallelism can result in the tasks of a single vertex being distributed across multiple thread groups, necessitating repeated loading of that vertex's neighbors. Systems such as TRUST [9] perform a vertex-parallel approach, while Tri-Core [10] employs an edge-parallel approach. Some systems, like G$^2$Miner [14], manually assign vertex-parallel or edge-parallel approaches depending on different patterns.

**Thread, Warp and Block Mapping**. Due to the hardware architecture of GPUs, GPU threads are typically assigned in three granular units for parallel execution: thread, warp, and block. A warp usually consists of 32 threads (in NVIDIA GPUs), which can trigger coalesced memory access when accessing adjacent data, thereby achieving higher bandwidth utilization, and a block usually comprises several warps. When processing a unit task (a vertex or an edge, depending on whether it is vertex-parallel or edge-parallel), we can use any of these three granular units. For high-degree vertices, using a block to collaboratively process the task can prevent the vertex's task from becoming a straggler. Conversely, for low-degree vertices, using an excessive number of threads does not result in performance gains. Therefore, bucketing vertices based on their degree and then selecting an appropriate parallel granularity (thread, warp or block) to process can achieve better load balancing.

## III. HYBRID SCHEDULING

In this section, we first describe our new optimization (re)introduced to existing optimizations, the traditional optimization is shown in Fig. 1 (b), and the parts that MERCURY changes are shown in Fig. 1 (c). We then propose how to combine these optimization variants in each category into a hybrid one to achieve robust performance.

Finally, we describe how our method scales the subgraph matching to multiple GPU cards.

### A. Hybrid Orientation Pruning

As described in § II, due to the symmetry in the pattern, we often constrain the IDs of the matched embeddings to satisfy certain restraints to reduce automorphisms. For example, in a triangle $\triangle(u, v, w)$, the symmetric breaking rules require that $id(u) > id(v) > id(w)$, where $id()$ represents the vertex identifier. By doing this, we can preemptively remove neighbors from the adjacency list that do not satisfy this constraint for each vertex, significantly reducing the computation load. In previous work, the *small-deg-first* strategy was commonly employed to facilitate symmetry breaking [8], [9], [14]. For example, an undirected edge $(u, v)$ will be removed from $u$'s neighbors if $degree(u) > degree(v)$. This method effectively mitigates issues of load balance arising from significant variations in vertex degrees. However, the vertex degree after orientation might be smaller than the number of threads within a thread group(block, warp), which can result in low parallel efficiency. The *large-deg-first* strategy with an opposite orientation addresses this issue. Therefore, we adopt a hybrid strategy that makes trade-offs between potential load balancing and parallel efficiency at the level of graph data.

### B. Hybrid Thread Mapping

Set operations typically involve two steps: *Load* the neighbor list (or hash table) into shared memory and *Search* the key in the list. We take triangle $\triangle(u, v, w)$ as an example, as shown in Fig. 2. *w.l.o.g.*, assuming a warp size of 3 and a block size of 6 (2 warps) for illustration. For the hash table method, we build the hash table using $N(u)$ in the *Load* step, then spend $O(1)$ time querying each vertex $w, w \in N(v)$ in the *Search* step. For the binary search method, we cache $u$'s neighbors list in the *Load* step, then spend $O(\log n)$ time for looking up vertex $w$ in the *Search* step. In traditional methods, a coarse-grained load balancing scheme is used, which involves binning vertices by their degree($N(u)$) and processing them using either one block or one warp based on their degree (depicted in Fig. 2 ❶). However, in real-world datasets, the degree of vertices in the *search* step is unlikely to exactly match the warp size or block size, leading to wasted computational resources or stragglers (depcited in Fig. 2 ❷).

Therefore, we adopt a hybrid thread mapping for decoupling set operations. To be more specific: (i) When loading the neighbor list in the *Load* step (depictetd in Fig. 2 ❸). a thread group(block, warp, thread) parallelism is employed according to the length of the search list. (ii) During the looking-up process in the *Search* step (depicted in Fig. 2 ❹), thread parallelism is used for fine-grained parallelism in the given thread group, *i.e.,* each thread is mapped to one edge of aggregated neighbors lists. This hybrid parallelization strategy not only effectively reduces the wastage of computational

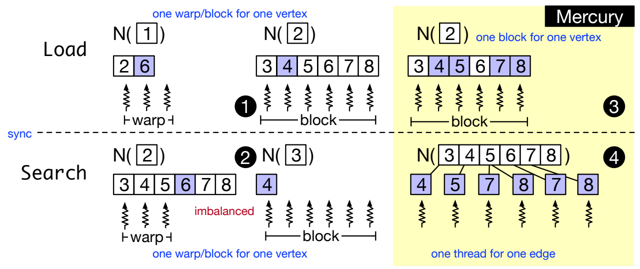

Fig. 2: Hybrid thread mapping strategy that devide the kernel into two phases with different parallelism-granularity.

resources but also makes the workload more balanced, thereby improving overall efficiency.

## C. *Hybrid Intersection Kernel*

In performing intersection operations, there are typically two processing methods: hash table and binary search, which were discussed in detail in § II. Currently, specialized systems for triangle counting, such as TRUST, TriCore, and H-INDEX, employ the hash table approach. This is because the search depth for triangle counting is relatively shallow and does not involve nested loops, allowing the hash table to free shared memory promptly upon obtaining a correct match. On the other hand, general subgraph matching systems like G$^2$Miner and GraphFold, often use the binary search method due to nested loops. In these cases, using a hash table would exhaust shared memory resources, making it inadequate for handling larger patterns.

Moreover, the performance behaviors of the intersection kernels between hash table and binary search also differ. As shown in Fig. 3, none of the two methods is superior in all scenarios. We can observe that the binary search method is more suitable for the small neighbor list, while the hash table method is more suitable for the large neighbor list. The reason behind this difference is that the hash table method accelerates lookup efficiency at the cost of building the hash table. When the number of neighbors is small, the overhead of building a hash table becomes significant. Instead of building the hash table, we can directly cache these low-degree vertex neighbors in GPU shared memory and perform the binary search.

Therefore, we utilize a hybrid intersection kernel by setting a threshold $th_d$. For vertices with a degree greater than $th_d$, the hash table method is employed, while for vertices with a degree less than it, the binary search method is used. In our experiments, we set $th_d$ to 384, as the performance of the hash table method is often better as depicted in Fig. 3.

## D. *Scale to multiple GPU cards*

Due to the large amount of intermediate results generated by subgraph matching, which far exceed the input graph data—for example, the number of triangles in Table I greatly surpasses the number of edges—we thus adopt a workload distribution approach [14], [16], [13]. As illustrated in the first loop of the query shown in Fig. 1 (a), it is necessary to evenly distribute the vertices in $|V|$ across different GPUs. Traditional methods use either vertex-parallel or edge-parallel approaches, dividing

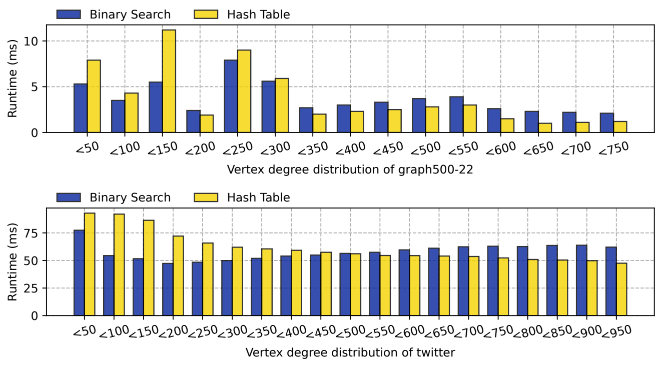

Fig. 3: Binary-search based intersection and hash-table based intersection.

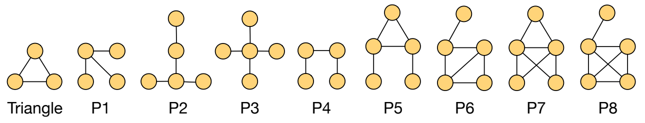

Fig. 4: Evaluated query patterns in this paper.

the vertices in $V$ into multiple chunks, each ensuring an equal number of vertices (vertex-parallel) or an equal number of edges (edge-parallel). Inspired by GSWITCH [18], we employ a hybrid partitioning method that ensures each chunk has an equal number of vertices and edges (sum of them). This share-nothing parallel approach offers good scalability and can achieve near-linear scalability (details in § IV).

## IV. EVALUATION

## A. *Experimental Setup*

We ran all experiments on a GPU cluster with 256 nodes, echo node is a Linux server equipped with a duel-socket AMD EPYC 7763 64-Core Processor and 4× 40 GB A100 GPUs. We compiled all the GPU programs using NVIDIA's nvcc compiler (version 11.7) and the -O3 flag. We tested MERCURY on 39 datasets (shown in Table I) and 9 patterns (shown in Fig. 4), on several graph mining tasks, including triangle counting, clique counting, and general subgraph matching. The datasets used are sourced from the official website of the Graph Challenge, including social networks, synthetic graphs, and Scientific graphs. All experiments of tested systems passed verification that produces the same results as a single-thread CPU-based standard implementation. To be fair, the measured results in all experiments ignored the IO time, output time, and preprocessing time (*e.g.,* graph orientation).

## B. *Triangle Counting*

**Comparison with Previous Graph Challenge Systems**. We compared MERCURY with recent GraphChallenge Triangle Counting champions: SMOG [16] (2023 winner), and H-INDEX [19] (2019 winner). We reproduced the results of SMOG and H-INDEX on our cluster, and the results are shown in Table I. MERCURY is 1.03× ∼ 19.7× (3.87× in average) times faster than SMOG and is 0.79× ∼ 21.6× (5.68 × in average) faster than H-INDEX.

TABLE I: MERCURY vs. SOTA subgraph matching systems on the triangle counting task.

| Runtime (ms) of Triangle Counting [Lower is better] | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Datasets** | **#V** | **#E** | **#△** | **MERCURY** | **TRUST** | **Sp.** | **G$^2$Miner** | **Sp.** | **SMOG** | **Sp.** | **H-Index** | **Sp.** |
| SNAP Datasets | | | | | | | | | | | | |
| amazon0312 | 400K | 2.3M | 3.6M | 0.98 | 1.07 | 1.09 | 1.09 | 1.11 | 2.22 | 2.26 | 1.50 | 1.53 |
| amazon0505 | 410K | 2.4M | 4M | 0.99 | 1.12 | 1.14 | 1.13 | 1.15 | 2.32 | 2.35 | 1.36 | 1.38 |
| amazon0601 | 403K | 2.4M | 4M | 1.01 | 1.81 | 1.80 | 1.14 | 1.13 | 2.33 | 2.32 | 1.45 | 1.45 |
| cit-Patents | 3.7M | 16.5M | 7.5M | 6.02 | 5.69 | 0.95 | 7.38 | 1.23 | 17.26 | 2.87 | 11.12 | 1.85 |
| flickrEdges | 106K | 2M | 108M | 3.50 | 5.48 | 1.57 | 3.65 | 1.04 | 5.38 | 1.54 | 5.32 | 1.52 |
| friendster | 65M | 1.8B | 4.1B | 1778 | 3170 | 1.78 | 1975 | 1.11 | 3945 | 2.22 | Err | - |
| roadNet-CA | 1.9M | 2.8M | 2.7M | 1.66 | 1.10 | 0.66 | 0.83 | 0.50 | 2.17 | 1.31 | 1.62 | 0.98 |
| roadNet-PA | 1.1M | 1.5M | 1.5M | 1.07 | 0.82 | 0.77 | 0.52 | 0.48 | 1.32 | 1.23 | 0.84 | 0.79 |
| livej | 4.8M | 43M | 286M | 24.19 | 26.22 | 1.08 | 26.32 | 1.09 | 54.00 | 2.23 | 41.72 | 1.72 |
| orkut | 3.1M | 117M | 628M | 105.74 | 132.8 | 1.26 | 116.7 | 1.10 | 233.5 | 2.21 | 205.1 | 1.94 |
| twitter20 | 21.3M | 265M | 17.3B | 551 | 1080 | 1.96 | 1190 | 2.16 | 2268 | 4.12 | 7262 | 13.18 |
| youtube | 7.1M | 57M | 103M | 12.58 | 17.27 | 1.37 | 21.63 | 1.72 | 46.67 | 3.71 | 38.26 | 3.04 |
| Synthetic Kronecker Datasets | | | | | | | | | | | | |
| 25-81-256-B1k | 548K | 2.1M | 2M | 1.11 | 0.62 | 0.55 | 1.33 | 1.20 | 1.99 | 1.79 | 1.99 | 1.79 |
| 25-81-256-B2k | 548K | 2.1M | 7 | 0.67 | 0.72 | 1.07 | 0.51 | 0.75 | 1.29 | 1.92 | 1.66 | 2.47 |
| 3-4-5-9-16-25-B1k | 530K | 11M | 35M | 13.64 | 20.30 | 1.49 | 22.26 | 1.63 | 28.11 | 2.06 | 53.13 | 3.89 |
| 3-4-5-9-16-25-B2k | 530K | 11M | 651 | 3.54 | 13.88 | 3.92 | 6.03 | 1.70 | 10.07 | 2.84 | 14.73 | 4.16 |
| 4-5-9-16-25-B1k | 133K | 1.6M | 3.5M | 2.12 | 2.59 | 1.22 | 1.73 | 0.81 | 2.19 | 1.03 | 2.21 | 1.04 |
| 4-5-9-16-25-B2k | 133K | 1.6M | 155 | 0.94 | 1.39 | 1.48 | 0.65 | 0.69 | 1.26 | 1.35 | 1.62 | 1.73 |
| 5-9-16-25-81-B1k | 2.1M | 28.6M | 66.7M | 35.77 | 52.04 | 1.45 | 65.85 | 1.84 | 61.10 | 1.71 | 261.57 | 7.31 |
| 5-9-16-25-81-B2k | 2.1M | 28.6M | 155 | 7.55 | 26.84 | 3.56 | 12.92 | 1.71 | 17.41 | 2.31 | 113.92 | 15.09 |
| 9-16-25-81-B1k | 362K | 2.6M | 4M | 1.90 | 2.49 | 1.31 | 2.33 | 1.23 | 2.39 | 1.26 | 4.51 | 2.38 |
| 9-16-25-81-B2k | 362K | 2.6M | 35 | 0.96 | 1.58 | 1.63 | 0.92 | 0.95 | 1.81 | 1.88 | 2.96 | 3.07 |
| MAWI Datasets | | | | | | | | | | | | |
| 201512012345 | 18.5M | 38M | 2 | 1.27 | 0.84 | 0.66 | 3.39 | 2.68 | 9.77 | 7.72 | 9.79 | 7.74 |
| 201512020000 | 40M | 74M | 2 | 1.81 | 1.09 | 0.60 | 6.55 | 3.61 | 18.78 | 10.36 | 18.93 | 10.44 |
| 201512020030 | 69M | 143M | 6 | 2.83 | 1.95 | 0.69 | 12.26 | 4.34 | 36.12 | 12.79 | 36.21 | 12.82 |
| 201512020130 | 128M | 270M | 10 | 3.63 | 2.46 | 0.68 | 22.82 | 6.28 | 68.14 | 18.77 | 58.90 | 16.22 |
| 201512020330 | 226M | 480M | 26 | 5.24 | 5.82 | 1.11 | 41.08 | 7.85 | 100.4 | 19.17 | 113.11 | 21.61 |
| Graph500 Datasets | | | | | | | | | | | | |
| scale18-ef16 | 262K | 4.2M | 82M | 4.06 | 7.30 | 1.80 | 4.68 | 1.15 | 9.08 | 2.24 | 10.32 | 2.54 |
| scale19-ef16 | 524K | 8.4M | 186M | 7.94 | 14.96 | 1.89 | 11.19 | 1.41 | 23.71 | 2.99 | 28.48 | 3.59 |
| scale20-ef16 | 1M | 16.8M | 419M | 21.96 | 43.18 | 1.97 | 33.76 | 1.54 | 67.57 | 3.08 | 107.4 | 4.89 |
| scale21-ef16 | 2.1M | 33.6M | 935M | 49.53 | 100.71 | 2.03 | 82.09 | 1.66 | 156.5 | 3.16 | 291.9 | 5.89 |
| scale22-ef16 | 4.2M | 67.1M | 2B | 108.2 | 217.3 | 2.01 | 176.5 | 1.63 | 407.3 | 3.77 | 861.2 | 7.96 |
| scale23-ef16 | 8.4M | 134M | 4.5B | 256.2 | 525.8 | 2.05 | 465.1 | 1.82 | 1011 | 3.95 | 2612 | 10.20 |
| scale24-ef16 | 16.8M | 268M | 9.9B | 542.2 | 1230 | 2.27 | 1113 | 2.05 | 2540 | 4.68 | 8052 | 14.85 |
| scale25-ef16 | 33.6M | 537M | 21.5B | 1294 | 3010 | 2.33 | 2761 | 2.13 | 6526 | 5.04 | 25290 | 19.55 |
| GenBank Datasets | | | | | | | | | | | | |
| P1a | 139M | 149M | 3.4K | 66.22 | 68.56 | 1.04 | 37.77 | 0.57 | 104.12 | 1.57 | 75.93 | 1.15 |
| U1a | 68M | 69M | 325 | 24.23 | 30.31 | 1.25 | 14.84 | 0.61 | 44.76 | 1.85 | 35.52 | 1.47 |
| V1r | 214M | 233M | 49 | 81.16 | 101.71 | 1.25 | 58.00 | 0.71 | 131.44 | 1.62 | 107.37 | 1.32 |
| V2a | 55M | 58.6M | 1.4K | 25.93 | 27.10 | 1.05 | 14.32 | 0.55 | 40.98 | 1.58 | 32.00 | 1.23 |

`Err` denotes that the system encountered an error. **Sp.** denotes the speedup of MERCURY over the corresponding system

**Comparison with Other SOTA Systems**. We Compared MERCURY with the state-of-the-art subgraph matching systems: TRUST [9] and G$^2$Miner [14]. The results are also shown in Table I. MERCURY is $0.55\times \sim 3.92\times$ ($1.46\times$ in average) faster than TRUST and $0.48\times \sim 7.85\times$ ($1.72\times$ in average) faster than G$^2$Miner.

**Scalability of Triangle Counting**. We then tested the scalability of our system by varying the number of GPUs from 1 to 1024 cards, on five large graphs, including *youtube*, *orkut*, *friendster*, *livejournal*, and *twitter* from SNAP datasets. The results are shown in Fig. 5. MERCURY scales well with the number of GPUs, and the speedup is almost linear with the number of GPUs. The speedup of MERCURY is $163\times \sim 630\times$ (average $380\times$) on 1024 GPUs compared to one GPU.

## C. Pattern Matching

**Comaprisons with G$^2$Miner**. G$^2$Miner [14] is a state-of-the-art subgraph matching system that uses a code generation technique to generate CUDA code for subgraph matching. We compared MERCURY with G$^2$Miner on 8 complex patterns (shown in Fig. 4). The speedup of MERCURY over G$^2$Miner is shown in Table 6. MERCURY is $1.11\times \sim 54.2\times$ ($9\times$ in average) faster than G$^2$Miner. This is because MERCURY uses a more balanced intersection kernel that significantly reduces the straggler and starvation.

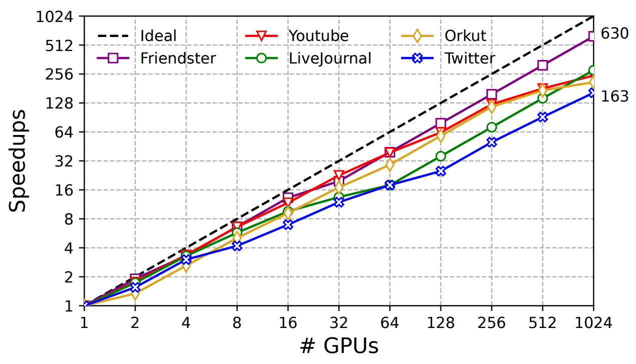

Fig. 5: The Strong Scalability of MERCURY in the triangle counting task.

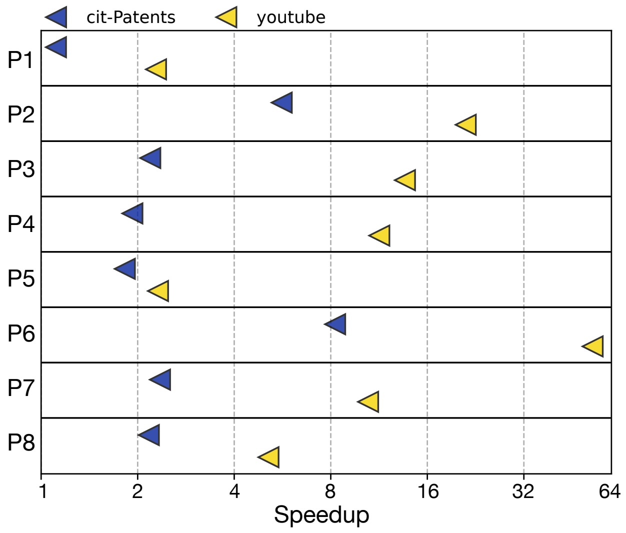

Fig. 6: The speedup of MERCURY over G$^2$Miner on subgraph matching tasks.

**Scalability of Generic Subgraph matching**. We then tested the scalability of our system by varying the number of GPUs from 1 to 1024 cards, on *youtube* graph, with 8 patterns. The results are shown in Fig. 7. MERCURY can achieve a speedup of 99 × ∼ 212× (average 136×) on 1024 GPUs compared to one GPU. The scalability of complex patterns is not as good as simple patterns, because the complex patterns have deeper search trees thus generating skewed workload distribution across GPUs, while our method mainly focuses on intra-GPU load balance.

## D. Ablation Study

We tested the effectiveness of each optimization in MERCURY by conducting an ablation study for triangle counting. `Baseline` means the original implementation without any optimization. `+OR` means we add adaptive orientation to make the degree of vertices larger than 32 (the size of warp), it can bring 32% speedup on average. `+LB` means we add the hybrid task mapping to avoid stragglers, it can bring an additional 62% speedup on average. `+BIN` means we add a binning strategy to use optimal intersection kernel for different vertices, it can further bring an additional 24% speedup on average.

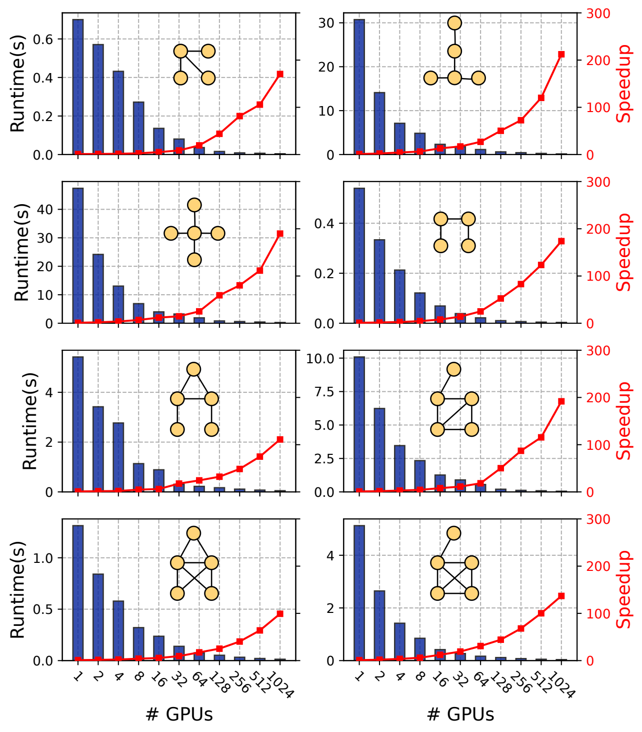

Fig. 7: The strong scalability of MERCURY in subgraph matching tasks (*Youbute* dataset) .

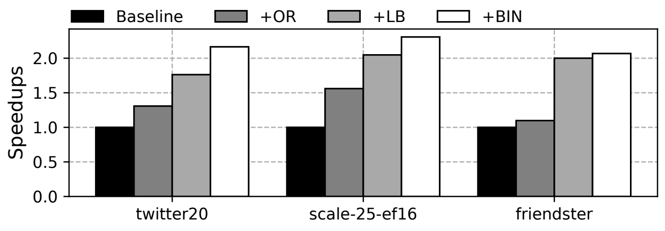

Fig. 8: The incremental speedups of MERCURY's optimizations.

## V. CONCLUSION

In this paper, we highlight that the issue of static optimizations can not rule all situations in graph pattern matching tasks. how input data in graph pattern matching tasks affects different optimization strategies. We first categorized the optimizations in graph pattern matching systems into three categories: (i) orientation pruning, (ii) intersection kernel, and (iii) thread mapping, and provided new optimization variants for each category. Based on these, we developed MERCURY, achieving high GPU utilization and load-balance. Currently, MERCURY's performance surpasses the state-of-the-art and previous GraphChallenge champions up to 21.5 × times on average across multiple datasets and applications.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments for their helpful suggestions. This work is supported by the National Key Research and Development Program of China (2023YFB3001900).

## REFERENCES

[1] N. Spirin and J. Han, “Survey on web spam detection: principles and algorithms,” *ACM SIGKDD Explorations Newsletter*, vol. 13, no. 2, pp. 50–64, 2012.

[2] A. Lancichinetti and S. Fortunato, “Community detection algorithms: A comparative analysis,” *Phys. Rev. E*, vol. 80, p. 056117, Nov 2009. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.80.056117

[3] Z. Guo, K. Guo, B. Nan, Y. Tian, R. G. Iyer, Y. Ma, O. Wiest, X. Zhang, W. Wang, C. Zhang, and N. V. Chawla, “Graph-based molecular representation learning,” in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, ser. IJCAI ’23, 2023. [Online]. Available: https://doi.org/10.24963/ijcai.2023/744

[4] S. Ranu and A. K. Singh, “Indexing and mining topological patterns for drug discovery,” in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 562–565. [Online]. Available: https://doi.org/10.1145/2247596.2247666

[5] L. Akoglu, H. Tong, and D. Koutra, “Graph based anomaly detection and description: a survey,” *Data mining and knowledge discovery*, vol. 29, pp. 626–688, 2015.

[6] C. C. Noble and D. J. Cook, “Graph-based anomaly detection,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 631–636.

[7] JEDEC, “High bandwidth memory (hbm) dram,” 2021. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

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

[9] 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 and Distributed Systems*, vol. 32, no. 11, pp. 2646–2660, 2021.

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

[11] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 678–689.

[12] L. Wang and J. D. Owens, “Fast gunrock subgraph matching (gsm) on GPUs,” 2020.

[13] Z. Lin, K. Meng, C. Shui, K. Zhang, J. Xiao, and G. Tan, “Exploiting fine-grained redundancy in set-centric graph pattern mining,” in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 175–187. [Online]. Available: https://doi.org/10.1145/3627535.3638507

[14] X. Chen and Arvind, “Efficient and scalable graph pattern mining on GPUs,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 857–877. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/chen

[15] D. Mawhirter and B. Wu, “Automine: Harmonizing high-level abstraction and high performance for graph mining,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 509–523. [Online]. Available: https://doi.org/10.1145/3341301.3359633

[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] N. S. Dasari, R. Desh, and Z. M, “pbitMCE: A bit-based approach for maximal clique enumeration on multicore processors,” in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 478–485.

[18] K. Meng, J. Li, G. Tan, and N. Sun, “A pattern based algorithmic autotuner for graph processing on gpus,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 201–213. [Online]. Available: https://doi.org/10.1145/3293883.3295716

[19] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu, “H-index: Hash-indexing for parallel triangle counting on gpus,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.