SIGMo: High-Throughput Batched Subgraph Isomorphism on GPUs for Molecular Matching

Antonio De Caro
antdecaro@unisa.it
University of Salerno
Salerno, Italy

Gennaro Cordasco
gcordasco@unisa.it
University of Salerno
Salerno, Italy

Federico Ficarelli
f.ficarelli@cineca.it
CINECA
Bologna, Italy

Biagio Cosenza
bcosenza@unisa.it
University of Salerno
Salerno, Italy

Abstract

Subgraph isomorphism is a fundamental graph problem with applications in diverse domains from biology to social network analysis. Of particular interest is molecular matching, which uses a subgraph isomorphism formulation for the drug discovery process. While subgraph isomorphism is known to be NP-complete and computationally expensive, in the molecular matching formulation a number of domain constraints allow for efficient implementations. This paper presents SIGMo, a high-throughput, portable subgraph isomorphism framework for GPUs, specifically designed for batch molecular matching. SIGMo takes advantage of the specific domain formulation to provide a more efficient filter-and-join strategy: the framework introduces a novel multi-level iterative filtering technique based on neighborhood signature encoding to efficiently prune candidates prior to a GPU-optimized join phase using a stack-based DFS traversal. The GPU implementation is written in SYCL, allowing portable execution on AMD, Intel, and NVIDIA GPUs. Our experimental evaluation on a large dataset from ZINC demonstrates up to 1470× speedup over state-of-the-art subgraph isomorphism frameworks, and achieves a throughput of 7.7 billion matches per second on a cluster with 256 GPUs.

CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**; • **Applied computing** → *Computational biology*.

Keywords

Subgraph Isomorphism, GPU, Molecular Matching

ACM Reference Format:
Antonio De Caro, Gennaro Cordasco, Federico Ficarelli, and Biagio Cosenza. 2025. SIGMo: High-Throughput Batched Subgraph Isomorphism on GPUs for Molecular Matching. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3712285.3759782

1 Introduction

Graph algorithms are a powerful abstraction for representing and modeling a wide variety of problems, and are indeed used in diverse domains such as biology [30], chemistry [51], and social network analysis [4]. High-performance implementations of graph algorithms are therefore extremely important to tackle the complexity of such analyses at scale, but efficient solutions are highly tailored to specific algorithms and the underlying architecture.

Graph isomorphism, which concerns determining whether two graphs are structurally identical (i.e., whether there exists a bijective mapping between their vertices that preserves edges), can be solved in quasipolynomial time [3]. In contrast, subgraph isomorphism, which asks whether a smaller graph (the *query graph* or pattern) exists as a subgraph within a larger graph (the *data graph* or target) with structure preserved under an injective mapping, is known to be NP-complete and, in general, considerably more computationally demanding.

Despite its computational cost, the subgraph isomorphism problem has broad applications across several scientific fields where the goal is to detect the presence of a known structure within a larger dataset. Examples of subgraph isomorphism applications range from computer vision [1, 57] to cheminformatics [15], and from graph databases [2] to machine learning [47].

In this paper, we focus on subgraph isomorphism for *molecular matching*: molecules and functional groups are represented by data and query graphs, modeled as undirected, cyclic, and labeled graphs where nodes represent atoms and edges represent chemical bonds. This formulation has particular relevance to computer-aided drug discovery process [22].

Despite the complexity of the underlying subgraph isomorphism formulation, the molecular matching problem offers a number of caveats that allow us for more efficient and domain-tailored algorithm implementations.

First, the problem is subject to domain constraints: a limited label set, low average degree, and high sparsity. Exploring such constraints allows advanced optimizations in the most computationally expensive part of the algorithm, for example, a more efficient filter in a filter-and-join strategy [52].

Second, while traditional graph analysis frameworks focus on scaling with the size of the input graph, in molecular matching we are more interested in scaling with the number of molecules we can process per second, i.e. high throughput for batch queries. This favors a different approach to parallelization, and in particular a different mapping to modern massively parallel GPU architectures.

This paper proposes a high-performance GPU implementation of batch subgraph isomorphism. The SYCL-based implementation is performance-portable and supports both node-to-node (*Find All*) and graph-to-graph (*Find First*) queries.

In summary, this paper makes the following contributions.

SC '25, St Louis, MO, USA

- A novel vertex filtering algorithm that iteratively refines candidate sets by progressively expanding each node's neighborhood, enabling early pruning of invalid matches;
- SIGMo, the first high-performance GPU framework for batched subgraph isomorphism, specifically designed for efficient molecular matching at scale, supporting both exhaustive enumeration of node-to-node matches (*Find All*) and graph-to-graph matches (*Find First*);
- A comprehensive experimental evaluation of SIGMo against state-of-the-art subgraph isomorphism frameworks, including performance comparisons across NVIDIA, AMD, and Intel GPU architectures, and a scalability study on a cluster of 256 GPUs.

2 Background

In cheminformatics, molecules are naturally represented as graphs, where atoms are vertices and atomic bonds are edges, both augmented with physical and chemical properties as shown in Figure 1.

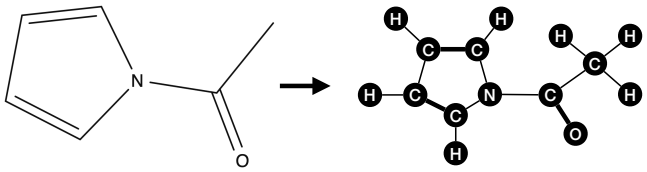


Figure 1: Graph representation of N-Acetylpyrrole molecule.

Rule-based cheminformatics methods rely on the enumeration of all isomorphisms between a query graph and a large number of data graphs. A common example of such methods is the enumeration of protonation states [45] where graph patterns are used to identify atoms with multiple proton configurations. Another common example is rule-based force fields, commonly used in tasks such as conformer generation and molecular dynamics. Computing higher-order parameters like bond torsional angles, dihedral angles, and bond lengths is typically done using quantum mechanics simulations. To avoid the high computational cost associated with quantum-level simulations, force fields are defined by precomputing such parameters for representative sets of functional groups (small molecular subgraphs) and collecting them in parameter tables. Each set of parameters is associated with an *atom type*, a label that enables the retrieval of quantum-level parameters for atoms based on their chemical environment (i.e., graph neighborhood). To perform atom typing, all valid subgraph isomorphisms between the input molecule (data graph) and all rules (query graphs) must be enumerated. All widely used force fields are based on atom typing: *biomolecular* (AMBER [38], CHARMM [54], OPLS-AA [26]), general organic (MMFF94 [19], UFF [39]), and specialized (CGenFF [53], Tripos [9]) force fields rely on isomorphism enumeration and are widely used in small molecule generation, molecular docking, and protein simulation. Rule-based force fields are the workhorses of cheminformatics.

Among other tasks such as conformer generation [33] and generative chemistry [42], the most challenging application of subgraph isomorphisms in terms of scale is searching for specific functional groups in large compound databases [5]. Compound databases are key assets for pharmaceutical companies, as they are typically curated and maintained as central components of drug discovery workflows [22]. Moreover, molecular databases comprising trillions of compounds are produced as outcomes of large-scale virtual screening campaigns [17].

2.1 Molecular Matching

In this work, we focus on the application of the subgraph isomorphism problem for the cheminformatics of small molecules—an area of particular relevance to computer-aided drug discovery [22]. To represent molecules and functional groups, both data and query graphs are modeled as undirected, cyclic, and labeled. Their vertices have degree-bounded by the maximum number of atomic bonds an element can form, according to its valence electron configuration and chemical context. Since drug discovery typically targets the chemical space of organic molecules, the distribution of vertex degrees cannot exceed 6 with an average value of approximately 4 (due to carbon atoms) [10]. Data graphs reflect the size of drug-like molecules, which usually consist of a few hundred atoms, with most drug molecules containing fewer than 200 atoms [28]. In contrast, the number of molecules processed during a virtual screening campaign can scale to the trillions [17]. Meanwhile, the number of patterns to be searched is fixed and relatively small, reaching up to a thousand only in specific fingerprinting tasks [31].

2.2 Problem Definition and Constraints

We use standard notation for graphs: a graph is a pair $G = (V_G, E_G)$ of sets of nodes V_G and edges E_G where each edge connects a pair of nodes, i.e., $E_G \subseteq V \times V$. By graph, we mean a simple, finite, undirected, connected graph. The order and size of G are denoted by $n = |V_G|$ and $m = |E_G|$.

For a set of nodes $X \subseteq V_G$, we denote $G[X]$ as the induced subgraph of G generated by X, that is, the graph whose node set is X and whose edge set consists of all the edges in E_G that have both endpoints in X. Given two nodes $u, v \in V_G$, we denote $d_G(u, v)$ as the distance between u and v in G. Moreover, for a node $v \in V_G$, we denote $N_G(v) = \{u \in V_G \mid (u, v) \in E_G\}$ as the neighborhood of v and $N_G^d = \{u \in V_G \mid u \neq v \wedge d_G(u, v) \leq d\}$ as the *neighborhood of radius d around v*. In the following, we omit the subscript G whenever the graph is clear from the context.

In this paper, we deal with node-labeled graphs where a set of labels $\mathcal{L}$ identifies some peculiarity of each node. Formally, a node-labeled graph is a triple $G = (V, E, L)$ where (V, E) defines the structure of the graph and $L : V \to \mathcal{L}$ is a mapping from the set of nodes V to the set of labels $\mathcal{L}$.

Definition 2.1. Let $G_D = (V_D, E_D, L_D)$ be a data graph and $G_Q = (V_Q, E_Q, L_Q)$ be a query graph. A subgraph $H = G_D[X]$ induced by $X \subseteq V_D$ is isomorphic to G_Q if there exists a bijection $f : V_Q \to X$ such that:

(1) for each $v \in V_Q$ we have $L_Q(v) = L_D(f(v))$
(2) if $(v, u) \in E_Q$ then $(f(v), f(u)) \in E_H$

The (1) indicates that the function f must preserve labels, while (2) ensures that all edges in the query graph are contained in the subgraph of the target graph induced by X.

We will consider the following problem.

> Node-Labeled Subgraph Matching (NLSM):
> **Input:** A data graph $G_D = (V_D, E_D, L_D)$ and a query graph $G_Q = (V_Q, E_Q, L_Q)$.
> **Output:** $\mathcal{X} = \{X \subseteq V_D \mid G_D[X] \text{ is isomorphic to } G_Q\}$.

This formulation sets the basis for our domain-aware subgraph matching strategies.

3 Molecular Matching Strategy

The NLSM problem is a fundamental problem with applications in various domains, including molecular matching for drug discovery where the graphs are characterized by:

- A limited label set, constrained by the chemical elements in the periodic table;
- A low average degree (typically ≤ 4), reflecting atomic valency constraints;
- High sparsity ($\geq 95\%$) [14];

Our approach follows the *filter-and-join* strategy [52], consisting of two main phases. In the filtering phase, the algorithm iteratively refines the set of candidate nodes for each node in the query graph, eliminating those that would lead to invalid results. Once filtering is complete, the joining phase begins, where candidate nodes are assembled into valid candidate chains, which are then combined and validated to produce the final solutions.

Filter. The filtering operation (see Algorithm 1) uses the concept of *node signature*, represented as an array of $|\mathcal{L}|$ integers, to evaluate compatibility between nodes. Note that node labels represent the intrinsic properties of graph nodes (e.g., atom type), whereas node signatures are derived features that encode the distribution of labels in their neighborhood.

The filtering process is conducted in multiple stages. At stage i, the signature of a node v is computed based on its neighborhood within a radius of i. As more stages are performed, additional candidates are filtered out, thereby simplifying the subsequent join operations. The goal is to find a trade-off between the number of stages and the number of resulting candidates.

Initially, the algorithm populates the set of candidates for each query node with data nodes that share the same label. In the first stage, it constructs a signature for each query and data node by counting, for each label, the number of neighboring nodes with that label.

To satisfy the conditions defined in Definition 2.1, the signature of a data node u must dominate the signature of the corresponding query node. Specifically, for each label $\ell \in \mathcal{L}$, the data node u must have at least as many neighbors with label ℓ as indicated in the query node's signature.

Subsequent stages are similar to the first one, but the signatures are computed over an increasingly extended neighborhood.

To build the signature of a node, we borrowed the idea of *n–view* from VSGM [25]. To identify the i-view of a particular node u, which corresponds to the neighborhood of radius i around u, we calculate the graph power G^i, defined as the graph that connects nodes in G if their distance is at most i. This is achieved by performing i BFS steps starting from node u.

Algorithm 1 Filtering process pseudocode.

```
1:  function Filter(G_Q = (V_Q, E_Q, L_Q), G_D = (V_D, E_D, L_D))
2:      C ← InitializeCandidates(G_Q, G_D)
3:      k ← 1
4:      repeat
5:          S_Q ← GenerateSignatures(G_Q, k)
6:          S_D ← GenerateSignatures(G_D, k)
7:          C ← RefineCandidates(G_Q, G_D, S_Q, S_D, C)
8:          k ← k + 1
9:      until k = s where s is the maximum amount of iterations
10:     return C
11: kernel GenerateSignatures(G = (V, E, L), k)
12:     ▷ S is the signatures matrix where S(v, l) denotes the number
        of occurrences of nodes having label l in the neighborhood of
        radius k around v
13:         ▷ R^k(v) is the number of nodes at distance k from v
14:     for all v ∈ V do                                   ▷ parallel for
15:         R^k(v) ← N^k(v) \ N^{k−1}(v)
16:         for all u ∈ R^k(v) do
17:             S(v, L(u)) ← S(v, L(u)) + 1
18:     return S
19: kernel RefineCandidates(G_Q, G_D, S_Q, S_D, C_prev)
20:     for all v_q ∈ V_Q do
21:         C(v_q) = ∅
22:     for all v_d ∈ V_D do                               ▷ parallel for
23:         for all v_q ∈ V_Q do
24:             if v_d ∈ C_prev(v_q) then
25:                 for all l ∈ ℒ do
26:                     if S_Q(v_q, l) ≤ S_D(v_d, l) then
27:                         C(v_q) ← C(v_q) ∪ {v_d}
28:     return C
29: kernel InitializeCandidates(G_Q, G_D)
30:         ▷ C is the candidate vector where C(v_q) is the set of
            candidates for v_q.
31:     for all v_q ∈ V_Q do
32:         C(v_q) = ∅
33:     for all v_d ∈ V_D do                               ▷ parallel for
34:         for all v_q ∈ V_Q do
35:             if L_Q(v_q) = L_D(v_d) then
36:                 C(v_q) ← C(v_q) ∪ {v_d}
37:     return C
```

The rationale behind this iterative approach is that the structural mismatches between the query graph and the data graph may not be immediately apparent at distance 1 but become evident when considering larger neighborhood contexts. By iteratively increasing the scope of the node's view, the algorithm systematically eliminates nodes that cannot be part of a valid mapping, reducing the search space before the more computationally expensive *join* phase.

During each refinement step, filtering operations are applied to further reduce the candidate set for each node. It is important to note that the filtering performed at iteration i must take into account the candidate set from iteration $i - 1$. Specifically, if a data

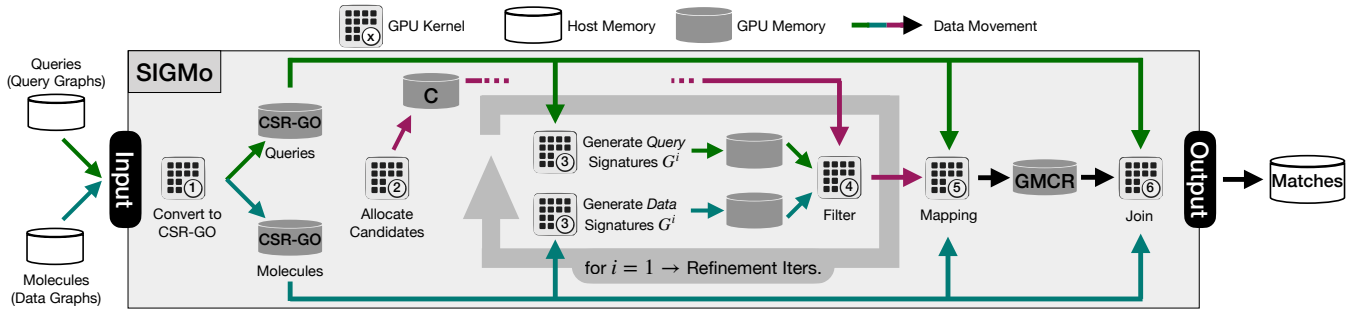


Figure 2: Framework Overview. SIGMo's pipeline includes six stages. It starts by converting input graphs into the CSR-GO ❶ format and initializing candidate sets ❷. The filtering phase then iteratively generates neighborhood-based signatures ❸ to prunes candidates ❹. After filtering, query graphs are mapped to data graphs ❺, and the join phase identifies valid subgraph matches ❻. Colored lines indicate inputs/outputs of each kernel throughout the pipeline.

node u_d is not a valid candidate for a query node u_q at iteration $i-1$, it cannot become a valid candidate at iteration i.

To filter multiple query and data graphs, we join all query graphs and all data graphs into two separate disconnected graphs.

Filter Complexity Analysis. To analyze the complexity of the *filter* algorithm (Alg. 1), let us break it down into its core components. The signature generation step performs a BFS starting from each node in the graph, which in total takes $O(n_d m_d)$ time, where n_d and m_d denote the number of nodes and edges in G_D respectively. The `InitializeCandidates` procedure takes $O(n_d n_q)$ time. Then, for each stage, the `RefineCandidates` procedure takes $O(n_d n_q |\mathcal{L}|)$ time. Hence, the overall complexity is $O(n_d m_d + k n_d n_q |\mathcal{L}|)$. Assuming that k and $|\mathcal{L}|$ are constants, the total cost of the filter algorithm is dominated by the time required to perform BFS traversals from each node in the data graph.

Join. The joining phase uses a backtracking approach over the pruned candidates to explore how they can be mapped to query nodes while preserving the query topology. During this process, edge labels are evaluated to prevent invalid matches.

4 SIGMo Implementation

In this section, we describe the implementation details of SIGMo. An overview of the framework pipeline is presented in Figure 2.

SIGMo is implemented using SYCL [18], a single-source, cross-platform abstraction layer that enables portable programming across heterogeneous hardware architectures. In contrast to most existing GPU-accelerated subgraph isomorphism frameworks—typically implemented in CUDA and thus limited to NVIDIA GPUs—SYCL allows SIGMo to target a broader range of devices, including NVIDIA, AMD, and Intel GPUs. This portability is particularly important given current hardware trends: as of November 2024, 7 out of the top 10 systems in the TOP500 list [50] are equipped with GPUs from vendors other than NVIDIA.

Throughout this section, we adopt the SYCL platform and memory model terminology. A *work-item* corresponds to a single GPU *thread* executing a kernel instance. A *work-group* is a one-, two-, or three-dimensional collection of work-items, analogous to a CUDA *block*. A *sub-group* represents a contiguous set of work-items that

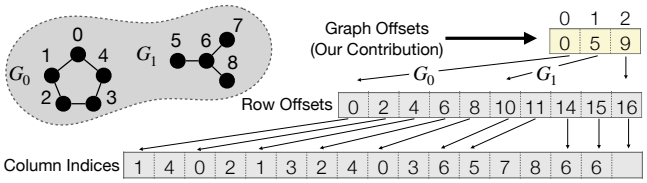


Figure 3: Illustration of the CSR-GO representation.

execute in Single Instruction Multiple Threads (SIMT) fashion; this concept is equivalent to a CUDA *warp* or an AMD *wavefront*. In terms of memory hierarchy, *local memory* refers to the *shared (local) memory* accessible to all work-items within a work-group, commonly used for low-latency communication and data reuse. In contrast, *private memory* denotes memory that is exclusive to a single work-item—analogous to *thread-local storage*.

4.1 CSR-GO Graph Representation

To represent both query and data graphs, we propose a data structure based on the classic Compressed Sparse Row (CSR) format [46], extended with an additional layer we term *graph offsets*. This representation, referred to as CSR-GO, is designed to handle disconnected graphs without losing information about connected components. Specifically, it introduces an auxiliary vector, graph offsets, whose length equals the number of graphs plus one. Each entry in this vector serves as a pointer mapping a segment of the row offsets array to a specific graph, working analogously to how row offsets map rows to adjacency lists. Figure 3 illustrates this representation.

Given a node ID, the corresponding graph can be efficiently determined via a binary search over the graph offsets array. This extension enables the storage and processing of multiple graphs within a unified structure, without duplicating metadata or sacrificing query performance. Moreover, it is particularly advantageous during the join operation, as described in Section 4.6. In our design, each work item is responsible for processing a single graph. As a result, the relevant range in the row offsets array can be efficiently retrieved by accessing only the graph offsets array.

4.2 Signature Representation

SIGMo vertex signatures are implemented as masked bitsets. Specifically, a 64-bit integer is partitioned into groups of bits, with each group corresponding to a particular vertex label. The number of supported labels is bounded by the set of elements in the periodic table, with a focus on those commonly found in organic molecules. However, element frequencies in organic compounds are highly skewed [36]; for example, hydrogen (H) and carbon (C) occur far more frequently than elements like silicon (Si).

To account for this imbalance, we apply a masking strategy that allocates more bits to frequently occurring labels (e.g., H and C), and fewer bits to rare ones (e.g., Si). This allows the signature to represent label counts more accurately while staying within the 64-bit constraint.

In cases where the count of a label exceeds the maximum representable value within its allocated bit group (i.e., overflow), the group remains unchanged. Despite this saturation, the resulting signature remains valid for filtering. This is because a data vertex is considered a valid candidate if, for each label, the count encoded in the query signature does not exceed that of the corresponding data signature.

4.3 Candidates Representation

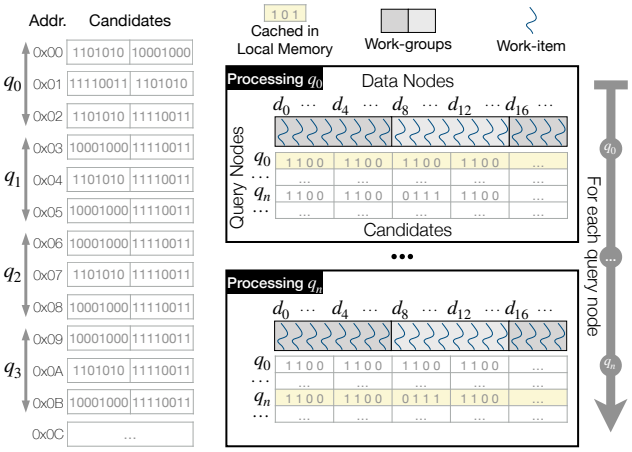
We represent the candidate set for each query node using a bitmap structure to facilitate insertion and removal operations. Specifically, we employ arrays of integers, where each bit set to 1 indicates a valid candidate data node. These bitmaps are stored in GPU memory in a contiguous, row-major layout—each row corresponding to a query node—to exploit coalesced memory access during filtering [49]. This layout ensures that threads within a sub-group access nearby memory locations, which helps optimize global memory bandwidth. Figure 4 illustrates this coalescing pattern. On modern GPUs, such access is considered coalesced, as long as the memory region is compact and properly aligned.

Updating the bitmap requires atomic operations to safely modify individual bits when multiple threads write concurrently. Contention is naturally limited because each integer in the bitmap covers only a small group of contiguous data nodes, and each thread is assigned to a single data node. As a result, atomic conflicts are limited to adjacent threads within the same sub-group that may access the same word. The granularity of the bitmap—determined by the number of data nodes represented per integer—can be tuned by adjusting the integer size. Aligning this granularity with the hardware's sub-group size can improve efficiency. However, if the integer size matches the sub-group size exactly, the memory controller may issue memory transactions containing only a single integer, leading to reduced throughput.

The candidate bitmaps are the most memory-intensive data structure in our pipeline. At peak usage, they consume up to 1 GB of GPU memory to represent 3,413 query nodes and 2,745,872 data nodes.

4.4 Filtering Candidates

The filtering process is divided into multiple refinement iterations, each separated by host-side synchronization. It consists of three



(a) Memory layout. **(b) Access pattern.**

Figure 4: Candidates representation.

distinct GPU kernels: *query signature refinement*, *data signature refinement*, and *candidates filtering*.

The *signature refinement kernels* assign one work-item per node—either in the query or data graph—and perform a BFS traversal starting from the assigned node. The depth of the BFS is determined by the current refinement iteration. To avoid restarting the BFS from scratch in each iteration, we cache the frontier after every step and reuse it as the starting point for the next iteration. Additionally, we maintain the set of nodes reached in each iteration to compute the difference from the previous step. This allows us to refine the signature using only newly discovered nodes.

The *candidate filtering kernel* assigns a single work-item to each data node as shown in Figure 4b. As discussed in the previous section, performance can degrade when the bitmap granularity aligns exactly with the hardware's sub-group size, due to inefficient memory coalescing. To address this, each work-item within a work-group prefetches the relevant bitmap integers into local memory before the filtering phase begins. This ensures efficient and coalesced access to memory across the entire work-group. During filtering, each work-item iterates over all query nodes to check whether its assigned data node is a valid candidate. For each query node, the work-item also iterates over a fixed set of labels, evaluating whether they satisfy the candidate validity conditions. In this filtering workload, increasing the work-group size can further improve performance, as memory bandwidth remains the primary bottleneck.

4.5 Mapping

Mapping is a crucial step to improve the performance of the join. In this step, each data graph is mapped only to the query graphs that are potential matches, discarding any query graph that contains nodes with zero candidates in that data graph. To efficiently store the mapping between data graphs and query graphs, we designed a Graph Mapping Compressed Representation (GMCR), which consists of two vectors: *data graph offsets* and *query graph indices*. The *data graph offsets* behaves similarly to the row offsets in the CSR

format, and stores the starting position of each data graph's entries in the *query graph indices*. The *query graph indices* contain the indices of all query graphs that potentially match a given data graph. In the GMCR, a boolean is designated for every query graph index to signify if a match occurred between that query graph and the respective data graph during the join phase.

The mapping phase consists of two kernels: the first kernel performs a prefix sum to compute the total size of the *query graph indices* vector, and to determine the offsets that populate the *data graph offsets* vector. To maintain consistency, the *data graph offsets* array is also updated on the host by performing an inclusive sum. The host then allocates the *query graph indices* and the boolean vectors, followed by the second kernel that populates the *query graph indices* vector.

4.6 Joining Partial Matches

In our evaluation, we considered both Depth-First Search (DFS) and Breadth-First Search (BFS) traversal strategies. While BFS generates multiple partial matches at each level—leading to an exponential increase in memory usage—DFS constructs only a single partial match per step, enabling more efficient memory usage. Additionally, DFS naturally aligns with backtracking approaches, as candidates can be evaluated sequentially along the traversal path.

Given that the query and data graphs we process are relatively small and exhibit tree-like structures—with treewidth not exceeding 2—both BFS and DFS produce comparable traversal orders. However, we adopt DFS due to its compatibility with backtracking and its superior memory efficiency.

To implement DFS-based backtracking in SIGMo, we account for the fact that GPU architectures do not support recursive function calls. Instead, we simulate recursion by maintaining an explicit stack data structure in private memory [56]. The maximum depth of this stack is bounded by the number of nodes in the query graph. Since our queries are small (no more than 30 nodes), we allocate a dedicated stack for each GPU work-item, allowing it to explore the search space without memory spillage.

In our execution model, each data graph is assigned to a work-group. The work-items within that group iterate over all valid query graphs, with each thread handling one query at a time. This thread-level parallelism is feasible because both the data and query graphs are small. By constraining the candidate set to only include nodes from the current data graph, each work-item can efficiently explore the full candidate space for a query without exceeding the available resources.

In contrast to the *filter* phase, which benefits from a larger work-group size to efficiently parse all candidates, the *join* phase performs better with a smaller work-group size, as the number of matching query graphs per data graph can vary significantly, leading to an under utilization of the GPU resources for large work-group sizes.

Although this approach may appear naive at first glance, we argue that it offers an effective balance between computational complexity and GPU resource management. Within the specific context of molecular graph matching—characterized by small graph sizes and low treewidth—this method achieves strong practical performance under real-world constraints.

5 Experimental Evaluation

We tested our approach on a dataset specifically designed to benchmark substructure searching algorithms in molecular graphs [16], from which we deleted single-atom patterns, resulting in 618 query graphs and 114,901 data graphs. This dataset was sourced from the ZINC database [24], which is currently the best source of commercially available molecular structures. We also used the whole ZINC dataset to assess the scalability of our framework.

The experiments were carried out on a system with dual Intel Xeon Gold 5218 CPUs, 192GB RAM, and an NVIDIA V100S GPU with 32GB VRAM. We compiled SIGMo with oneAPI v2024.2.0 [23] compiler and CUDA v555.42 drivers.

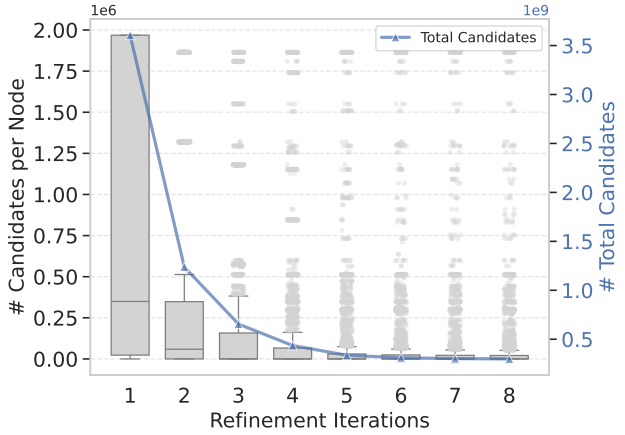


Figure 5: Summary of the distribution of candidate set sizes for each refinement iteration. The box represents the distribution of the candidates set sizes for each node and aligns with the left axis, whereas the line indicates the total number of candidates, aligning with the right axis.

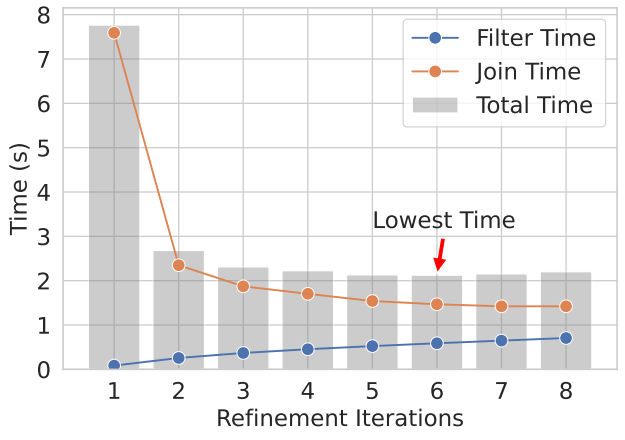


Figure 6: Comparison of filter and join time per each refinement iteration on the entire dataset.

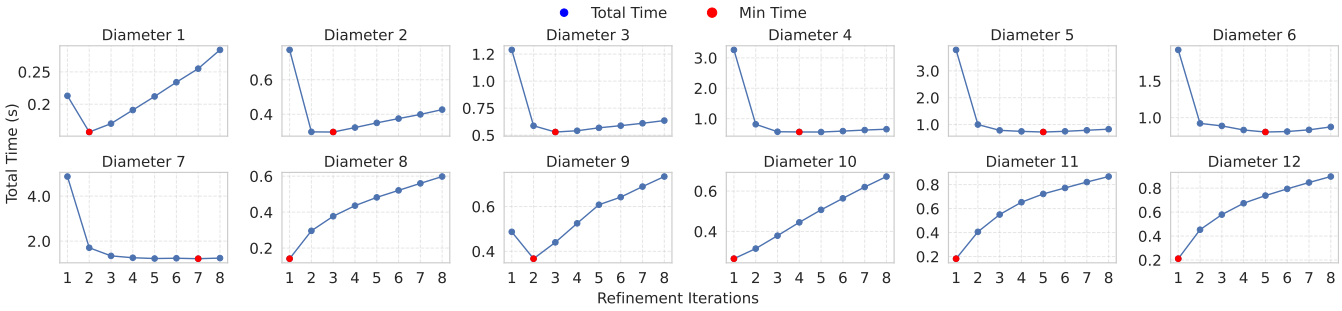


Figure 7: Total execution time of SIGMo across refinement iterations, grouped by query graph diameter.

5.1 Assessing SIGMo

In this section, we evaluate the performance of SIGMo. It is important to clarify that when we refer to refinement iteration i, it means that each node has visibility over its neighbors up to distance $i - 1$. For instance, refinement iteration 1 implies that each node is only aware of its own label, with no neighborhood context.

5.1.1 Candidate Sets Pruning. Each refinement iteration begins with the signature refinement of both query and data graph nodes. While some node signatures converge earlier than others, the overhead of continuing to refine already-converged nodes is negligible relative to the overall computational cost. In practice, the total time required to perform signature refinement across all query and data nodes does not exceed 10 milliseconds, even in the largest datasets.

Figure 5 illustrates the distribution of the candidate set sizes across query nodes (represented as box plots) and the total number of candidates (shown as a line). A significant reduction in candidate sets is observed after the first iteration, highlighting the effectiveness of early pruning. Beginning around iteration 6, the total number of candidates plateaus, indicating that most query graphs have reached convergence and no longer benefit from further refinement.

Despite this convergence, outliers persist across iterations, particularly in the earlier stages. These outliers are attributed to query patterns that correspond to frequent molecular substructures, which are more likely to occur across a wide range of molecules, and thus resist pruning. As shown in Figure 5, these outliers do reduce their candidate sets in later iterations—when they gain a broader view of their neighborhood—but they still retain a relatively large number of candidates compared to the rest.

5.1.2 Filter vs. Join. Figure 6 presents a comparison between the execution times of the filter and join phases across different refinement iteration counts. The results reveal a turning point: beyond a certain number of refinement iterations, the cost of additional filtering outweighs the performance gains achieved during the join phase. In other words, excessive refinement may reduce the candidate set further, but at the expense of increased overhead that negates the benefits in subsequent stages.

This observation is supported by Figure 5, which shows that the total number of candidates begins to plateau after iteration 6. Beyond this point, only a marginal number of additional candidates are eliminated, offering decreasing returns in terms of join phase speedup, hence resulting in a higher overall runtime.

It is important to note that this optimal refinement depth may vary depending on the diameter of the query graphs. Datasets containing query graphs with larger diameters may require more iterations before convergence is reached, as a broader neighborhood view becomes necessary to effectively prune candidates. To investigate this, we grouped the query graphs based on their diameters and balanced the groups to contain the same number of graphs. Figure 7 illustrates the total execution time of SIGMo for these grouped query graphs. As the diameter increases, we observe that the execution time curves shift to the right, indicating that the best number of refinement iterations occurs later. This indicates that graphs with larger diameters require more refinement steps. Anomalies appear in the cases with diameters 8, 10, 11, and 12, where the execution exhibits irregular behavior. These query graphs did not produce matches because in each case at least one node had zero candidates from the first iteration. This led to null join operations, as the mapping phase failed to associate any query graph with a corresponding data graph. A similar behavior is observed in the group of query graphs with a diameter of 9, where the GMCR determines that no matches are possible only starting from the second iteration.

5.1.3 Resources Utilization. On the evaluated dataset comprising 618 query graphs and 114,901 data graphs—for a total of 3,413 query nodes and 2,745,872 data nodes—SIGMo occupies approximately 1 GB of memory. In particular, 80% of the memory footprint is attributed to the bitset-based representation of the candidate sets. The candidate size can be determined in advance by considering $|V_Q| \times |V_D|/8$ bytes. The data graphs account for approximately 64 MB of memory usage, while the query graphs require only 90 KB, both represented in the CSR-GO format. Additionally, the signature representations for both query and data nodes collectively consume around 128 MB.

Figure 8 shows the percentage of GPU occupancy during SIGMo execution in six refinement iterations, profiled through NVIDIA DCGM which defines GPU occupancy as *the fraction of resident warps on a multiprocessor, relative to the maximum number of concurrent warps supported on a multiprocessor* [34]. The test was performed on an NVIDIA V100S GPU. The results reveal that the filtering phase reaches peak GPU utilization. The observed drops

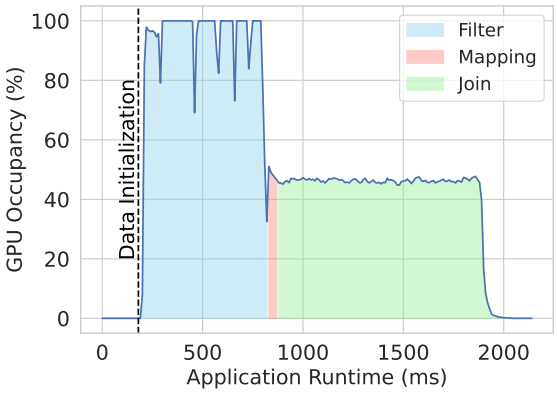


Figure 8: Profiling of the NVIDIA V100S GPU occupancy during the SIGMo runtime with six refinement iterations.

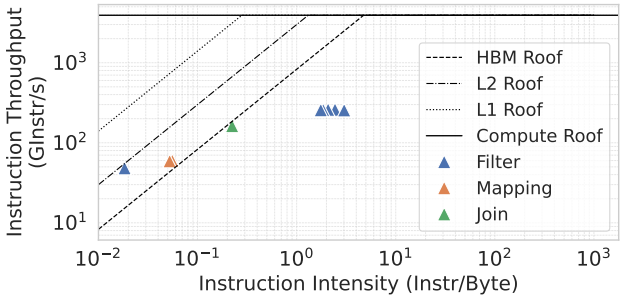
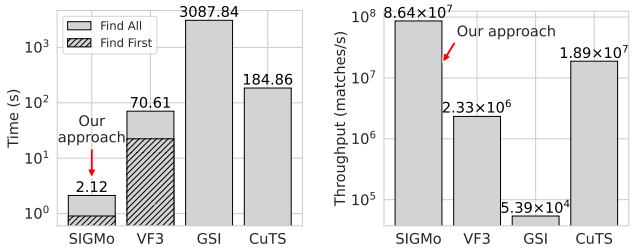


Figure 9: Instruction Roofline of SIGMo execution with six refinement iterations on NVIDIA V100S.

in occupancy are primarily attributed to host-side synchronization overhead, as is evident by the presence of six distinct peaks corresponding to the filter phase. The mapping phase is relatively brief, lasting approximately 50 milliseconds. This short duration contributes to the observed GPU occupancy, which ranges between 47% and 55%. In contrast, the join phase exhibits a more stable occupancy of around 48%, mainly due to memory bottlenecks arising from the irregular access patterns required to read the query and data graphs. This behavior is evident in Figure 9, which presents the *Instruction Roofline Model* (IRM) [12]—a more suitable tool for our use case compared to the standard Roofline Model [29]. The first filter kernel considers the neighborhood at distance 0, which means that only the label is evaluated, motivating the low instruction intensity.

The underutilization of the join phase observed in Figure 8 is mainly attributable to warp-level divergence: different threads process query graphs of varying size and complexity, which leads to heterogeneous control paths and reduced occupancy. While alternative designs such as assigning one query per sub-group can reduce divergence, they also lower parallelism and increase memory contention, resulting in an increased overall execution time.



(a) Execution time (the lower is better). **(b) Throughput (the higher is better).**

Figure 10: Comparison of SIGMo with other CPU and GPU state-of-the-art subgraph isomorphism frameworks.

5.2 State-of-the-Art Comparison

We evaluated our framework against three leading frameworks, namely *VF3* [7], *GSI* [61], and *cuTS* [58]. nvcc v12.3 compiled GSI and cuTS, while VF3 was compiled using g++ 11.4.0.

In all the experiments we did not consider the time to allocate and initialize data structures. To run the experiments on VF3, GSI, and cuTS, we merged the data graphs into a single disconnected graph and tested queries individually. VF3 appears to be a better solution for matching several queries on a large set of data graphs compared to GSI and cuTS. In addition, GSI ran out of memory on the largest query graphs (on graphs with more than 20 nodes). Figure 10a shows the comparison of the execution time to find matches. Both SIGMo and VF3 support the early stopping when finding a match between a query graph and a data graph, while GSI and cuTS do not. We achieve a speedup of 33.6× compared to VF3, 1470.4× compared to GSI, and 88× compared to cuTS.

Figure 10b shows the throughput, defined as the number of matches per second. To calculate the throughput, we considered for both SIGMo and VF3 the time required to find all the matches. The cuTS framework does not support labels, leading to a higher number of matches for a single query graph.

In summary, the performance difference is mainly attributed to SIGMo's design: unlike previous frameworks optimized for parallelism on a single large graph, SIGMo targets high-throughput execution on batches of labeled sparse graphs. Its iterative filtering prunes the candidate space, CSR-GO and GMCR ensure memory-efficient storage and mapping, and the stack-based DFS join enables thread-local backtracking—together accounting for the large performance gains in Figure 10.

5.3 Performance Portability

Assessing performance portability is inherently challenging, particularly when evaluating a novel solution like SIGMo that has no direct counterpart or baseline [37]. In this section, we provide insights into how SIGMo performs across different hardware platforms. We evaluated SIGMo on three different systems, each equipped with a different GPU architecture: an NVIDIA V100S, an AMD MI100, and an Intel Max 1100. Compilation was performed using oneAPI v2024.2 across all platforms to ensure consistency.

Table 1: SIGMo configuration on three hardware platforms.

GPU	Candidates bitmap integer	Filter Work-group size	Join Work-group size
NVIDIA V100S	32 bit	1024	128
AMD MI100	64 bit	512	64
Intel Max 1100	32 bit	512	32

Figure 11 presents the execution time across two main computation phases—filter and join—and the overall execution time for multiple refinement iterations, while Table 1 reports the best configuration parameters for SIGMo identified through manual tuning.

Among the evaluated platforms, the AMD MI100 consistently delivers the fastest execution times, reaching a minimum of 1.70 seconds at five refinement iterations, compared to 2.12 seconds for the NVIDIA V100S at six iterations and 2.65 seconds for the Intel Max 1100 at two iterations. In contrast, the Intel Max 1100 exhibits the highest total runtimes, driven primarily by the elevated cost of the Filter phase. On this device, the overhead of additional refinement iterations outweighs the benefits of further candidate reduction, making extra iterations less advantageous.

To better contextualize performance variations, we break down the computation into its two main phases: Filter and Join. All metrics discussed below were obtained using *VTune* (Intel), *Nsight Compute* (NVIDIA), and *Rocprof* (AMD). The Filter phase is more compute-intensive with a low memory footprint (<1.2GB). This puts more pressure on the compute units rather than memory. In this setting, architectural differences naturally emerge: the Intel GPU offers significantly lower peak compute performance (22 TFLOPS) compared to AMD MI100 (180 TFLOPS) and NVIDIA V100S (130 TFLOPS). Despite this difference, all GPUs achieve over 93% of their sustained peak compute throughput during the Filter phase with two refinement iterations. With a single refinement iteration, the Filter phase becomes memory-bound, as shown in the Roofline plot (Figure 9). In this case, Intel's higher memory bandwidth enables it to outperform the other devices. The Join phase initially incurs substantial memory traffic (16 GB) due to the large number of candidates stored in the GMCR. This results in memory-dominated behavior during the first iteration, again favoring the Intel Max GPU. As candidates are pruned, memory access becomes more selective, and the workload shifts toward compute. Notably, L2 cache hit rates exceed 90% across all GPUs and iterations while occupancy remains around 50%. Additionally, during the Join phase with a single refinement iteration–when each query graph retains many candidates–AMD shows the highest sensitivity to control-flow divergence, due to its larger wavefront size (64 threads vs. 32 for NVIDIA and 16 for Intel). The wider execution group increases the chance of divergence within a wavefront, reducing execution efficiency. This effect is no longer observed with additional refinement iterations. Overall, hardware profiling reveals consistent behavior across all platforms starting from the second refinement iteration, indicating that observed performance differences reflect inherent hardware capabilities.

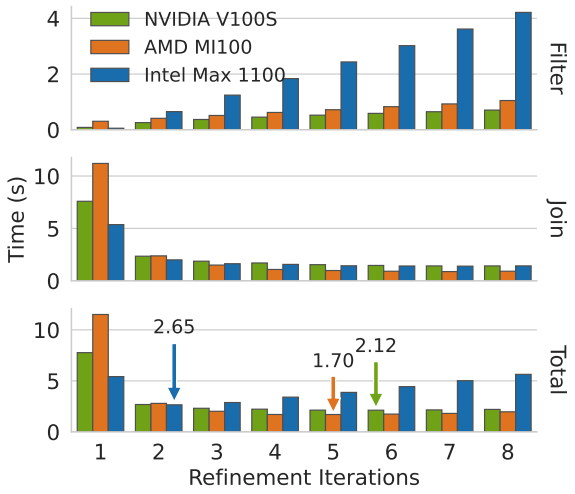


Figure 11: Filter, join, and total execution times of SIGMo on NVIDIA V100S, AMD MI100, and Intel Max 1100 GPUs. The total time includes an arrow indicating the fastest execution for each GPU.

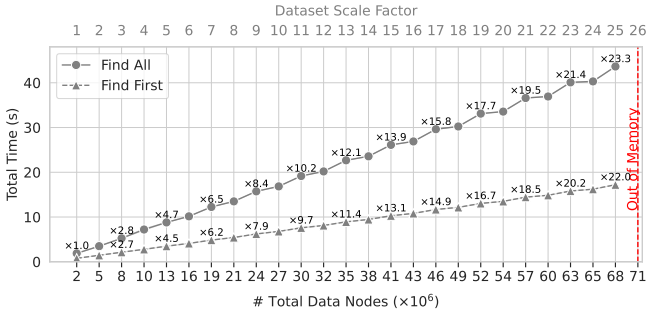


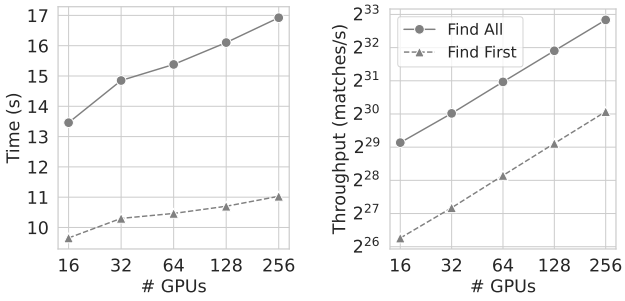
Figure 12: Single-GPU scalability of SIGMo in both 'Find All' and 'Find First' modes. The plot shows how the performance scale by increasing the dataset size. The bottom x-axis represents the size of the data graphs, while the top x-axis indicates the corresponding dataset scale factors. Numbers along each line denote the relative execution time compared to the baseline (first execution).

The relatively small performance gap between the three architectures provides additional evidence of the efficiency and portability of our approach, demonstrating that SIGMo can achieve competitive performance even on diverse hardware.

5.4 Scalability Evaluation

We evaluated the weak scaling of SIGMo on single and multiple GPUs.

5.4.1 Single GPU Scalability. We evaluated the scalability of our framework on a single GPU. Figure 12 illustrates how SIGMo performance scales as the number of data graphs increases up to the maximum available memory of the GPU, while the number of



(a) Execution time (the lower is better). **(b) Throughput (the higher is better).**

Figure 13: Execution of SIGMo on a multi-node environment with up to 256 NVIDIA A100 GPUs.

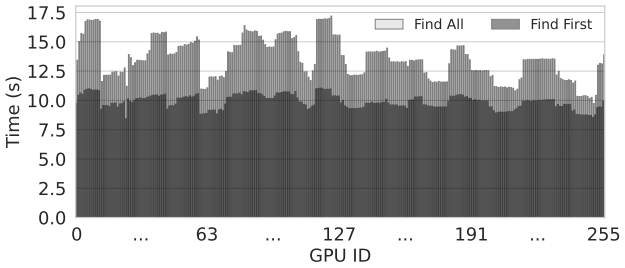


Figure 14: Runtime of each MPI process on 256 GPUs.

query graphs remains constant. Overall, the framework demonstrates good scalability with input size, exhibiting sublinear growth in execution time. This trend is especially clear in the *Find First* execution. In contrast, the *Find All* execution displays more variability and more pronounced increases in runtime at higher scale factors. However, this is acceptable because we assign a different data graph to each work-group. As a result, when all available compute units are saturated, SYCL schedules these executions into multiple join kernels. This explains the overhead observed, for example, when increasing the scale factor from 16 to 17.

5.4.2 Multi Node Scalability. We evaluated the performance of SIGMo on an HPC cluster, where each node is equipped with four NVIDIA A100 GPUs. The experiments were carried out using molecules extracted from the ZINC dataset [48], along with a fixed set of 389 queries. For inter-node communication, we used Intel MPI v2021.11. Figure 13 reports the median results of five executions performed with 16, 32, 64, 128, and 256 GPUs, each running six refinement iterations. The plot demonstrates that our framework scales efficiently across the cluster, exhibiting linear performance gains in log-log space as the number of nodes increases.

We used static partitioning on the ZINC dataset, assigning 500,000 molecules to each GPU. Consequently, increasing the number of nodes led to a proportional increase in the total number of molecules processed from the dataset. Figure 14 illustrates the runtime of each MPI process, where each process is mapped to a single GPU in the 256-GPU configuration. Due to the static partitioning strategy, variations in execution time are observed due to the different number of candidates produced, reflecting differences in the molecular workloads assigned to each process [43]. Although more adaptive load-balancing approaches have been shown to improve scalability [27], the observed runtime variability remains low, with a coefficient of variation of only 4% in the *Find First* execution and 8% in the *Find All* execution.

At peak scale, SIGMo successfully processed up to 128 million molecules in about 17 seconds, producing 129,575 billion total matches in the *Find All* execution and achieving a peak throughput of up to 7.7 billion matches per second.

6 Related Work

Subgraph isomorphism is a well-known NP-complete problem with extensive research across CPU and GPU platforms. The early foundational work by Ullmann [52] laid the theoretical foundations, introducing a backtracking algorithm with pruning strategies. This was followed by more advanced techniques such as VF2 [11], and its successors VF2Plus[8] and VF3 [7], which introduced more advanced state-space search techniques with improved performance on biological graphs [16], making them widely used on CPU architectures. Several CPU-based subgraph isomorphism algorithms have shown strong performance across various benchmarks. RI and its extension RI-DS [6] use recursive search and degree sequence filtering to efficiently prune the candidate space, particularly in sparse graphs. The Glasgow Subgraph Solver [32] applies constraint programming techniques combined with bitset-based data structures. TurboISO [21] introduces a neighborhood label frequency index and region exploration for fast matching over large graphs. QuickSI [44] focuses on minimizing verification time using a heuristic-driven matching order, making it one of the first scalable solutions for labeled graphs. These frameworks, while highly optimized for single-query execution, are not designed for large batched molecular matching, which is the primary focus of SIGMo.

Although CPU-based algorithms remain effective, they struggle to scale to high-throughput workloads. This has led to increased interest in GPU-based frameworks that exploit massive parallelism. Among these, cuTS [58], GSI [61], PARSEC [13], and STMatch [56] are notable. CuTS uses a trie-based structure and performs well in distributed multi-GPU environments, Parsec accelerates subgraph enumeration using parallel traversal and candidate expansion strategies, while STMatch accelerates subgraph matching on GPUs using stack-based loop optimizations to replace recursive DFS, reducing divergence and improving efficiency for individual pattern matches. These frameworks lack support for labeled graphs, which limits their utility in molecular domains, where labels represent meaningful constraints for matching. GSI and its scalable extension SGSI [60] leverage parallelism on GPU architectures, but suffer from high memory overhead, especially when processing large query graphs. DGSM [20] also implements GPU-accelerated subgraph matching using depth-first search strategies using a backtracking unrolling strategy. However, it does not target specific constraints of batched molecular matching, and their scalability is often limited to smaller datasets or single query workloads. Our method draws inspiration from VSGM [25], which introduces a view-based approach to filter

Table 2: Comparison against the state of the art.

	Domain-specific	GPU Offload	Batched Matching	Exact Matching
O'Boyle et al. [35]	✓	✗	✗	✗
Carletti et al [7]	✗	✗	✗	✓
Xiang et al. [58]	✗	CUDA	✗	✓
Zeng et al. [60, 61]	✗	CUDA	✗	✓
Our work	✓	Heterog.[1]	✓	✓

candidate nodes by examining multi-hop neighborhoods. Although VSGM achieves strong filtering efficiency, it operates on single query-data graph pairs and does not address batching or memory optimization for large-scale molecular datasets. SIGMo builds upon this idea by introducing iterative signature refinement [40], allowing highly selective filtering that reduces the search space before the join phase. This is especially critical for molecular matching, where node labels and neighborhood context carry significant meaning.

Various techniques for molecular matching circumvent the need for subgraph isomorphism, opting instead for fingerprint-based algorithms [40], canonical SMARTS/SMILES evaluation [35], and molecular embeddings learned through graph neural networks (GNNs) [41, 55, 59]. Despite their efficiency, these methods are inherently approximate and can produce not only false positives, but also false negatives—potentially missing relevant molecular matches.

Table 2 summarizes the contributions of SIGMo in relation to some of the state-of-the-art frameworks. *In contrast to these works, SIGMo is the first portable high-performance subgraph isomorphism GPU framework for molecular matching, designed to query a large dataset of molecules simultaneously in a batched fashion.*

7 Conclusion

In this study, we presented SIGMo, the first *portable and high-throughput* GPU subgraph isomorphism framework tailored for the molecular matching problem, which exploits GPU parallelism to match multiple queries on multiple data graphs simultaneously. We also proposed a novel filter strategy that is designed to prune candidates in labeled graphs through inspection of neighborhood constraints. Although we focused on molecular matching, the filter strategy is broadly applicable to labeled sparse graphs and can also be applied in domains such as malware detection and graph database queries, reinforcing its relevance beyond this specific application.

Experimental results show that SIGMo significantly outperforms the current state-of-the-art GPU solution, achieving speedups of up to 1470.4× and surpassing the leading CPU-based solution by up to 33.6×. Furthermore, SIGMo demonstrates excellent scalability, reaching a peak throughput of 7.7 billion matches per second on a cluster equipped with 256 GPUs. As a next step, we plan to extend SIGMo to support wildcard atoms and bonds, which are used in cheminformatics to express flexible or partially specified substructures.

[1]Heterog. refers to support for heterogeneous backends such as CUDA, HIP, LevelZero, and OpenCL.

References

[1] M. A. Abdulrahim and M. Misra. 1998. A Graph Isomorphism Algorithm for Object Recognition. *Pattern Analysis and Applications* 1, 3 (1998), 189–201. doi:10.1007/BF01259368

[2] Merve Asiler and Adnan Yazıcı. 2018. *BB-Graph: A Subgraph Isomorphism Algorithm for Efficiently Querying Big Graph Databases.* doi:10.48550/arXiv.1706.06654 arXiv:1706.06654 [cs]

[3] László Babai. 2016. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016.* ACM, 684–697. doi:10.1145/2897518.2897542

[4] John A Barnes and Frank Harary. 1983. Graph theory in network analysis. *Social networks* 5, 2 (1983), 235–244.

[5] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A Subgraph Isomorphism Algorithm and Its Application to Biochemical Data. *BMC Bioinformatics* 14, S7 (2013), S13. doi:10.1186/1471-2105-14-S7-S13

[6] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14 (2013), 1–13.

[7] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Introducing VF3: A new algorithm for subgraph isomorphism. In *Graph-Based Representations in Pattern Recognition: 11th IAPR-TC-15 International Workshop, GbRPR 2017, Anacapri, Italy, May 16–18, 2017, Proceedings 11.* Springer, 128–139.

[8] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. 2015. VF2 Plus: An improved version of VF2 for biological graphs. In *Graph-Based Representations in Pattern Recognition: 10th IAPR-TC-15 International Workshop, GbRPR 2015, Beijing, China, May 13-15, 2015. Proceedings 10.* Springer, 168–177.

[9] Matthew Clark, Richard D. Cramer, and Nicole Van Opdenbosch. 1989. Validation of the General Purpose Tripos 5.2 Force Field. *Journal of Computational Chemistry* 10, 8 (1989), 982–1012. doi:10.1002/jcc.540100804

[10] Jonathan Clayden, Nick Greeves, and Stuart Warren. 2012. *Organic Chemistry* (2nd ed ed.). Oxford university press.

[11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[12] Nan Ding and Samuel Williams. 2019. An Instruction Roofline Model for GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* IEEE, 7–18. doi:10.1109/PMBS49563.2019.00007

[13] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. PARSEC: Parallel subgraph enumeration in CUDA. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 168–178.

[14] Yuanqi Du, Shiyu Wang, Xiaojie Guo, Hengning Cao, Shujie Hu, Junji Jiang, Aishwarya Varala, Abhinav Angirekula, and Liang Zhao. 2021. GraphGT: Machine Learning Datasets for Graph Generation and Transformation. In *NeurIPS 2021.*

[15] Hans-Christian Ehrlich and Matthias Rarey. 2011. Maximum Common Subgraph Isomorphism Algorithms and Their Applications in Molecular Science: A Review. *WIREs Computational Molecular Science* 1, 1 (2011), 68–79. doi:10.1002/wcms.5

[16] Hans-Christian Ehrlich and Matthias Rarey. 2012. Systematic benchmark of substructure search in molecular graphs-From Ullmann to VF2. *Journal of cheminformatics* 4 (2012), 1–17.

[17] Davide Gadioli, Emanuele Vitali, Federico Ficarelli, Chiara Latini, Candida Manelfi, Carmine Talarico, Cristina Silvano, Carlo Cavazzoni, Gianluca Palermo, and Andrea Rosario Beccari. 2022. EXSCALATE: An Extreme-Scale Virtual Screening Platform for Drug Discovery Targeting Polypharmacology to Fight SARS-CoV-2. *IEEE Transactions on Emerging Topics in Computing* 11, 1 (2022), 1–12. doi:10.1109/TETC.2022.3187134

[18] The Khronos® SYCL™ Working Group. 29-03-2023. *SYCL 2020 Specification (revision 8) — registry.khronos.org.* https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[19] Thomas A. Halgren. 1996. Merck Molecular Force Field. I. Basis, Form, Scope, Parameterization, and Performance of MMFF94. *Journal of Computational Chemistry* 17, 5–6 (1996), 490–519. doi:10.1002/(SICI)1096-987X(199604)17:5/6<490::AID-JCC1>3.0.CO;2-P

[20] Wei Han, Connor Holmes, and Bo Wu. 2022. DGSM: A GPU-Based Subgraph Isomorphism framework with DFS exploration. In *2022 IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop (RSDHA).* IEEE, 1–11.

[21] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. 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.* ACM, 337–348.

[22] Jp Hughes, S Rees, Sb Kalindjian, and Kl Philpott. 2011. Principles of Early Drug Discovery. *British Journal of Pharmacology* 162, 6 (2011), 1239–1249. doi:10.1111/j.1476-5381.2010.01127.x

[23] Intel®. 2024. *oneAPI Base Toolkit.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html

[24] John J Irwin and Brian K Shoichet. 2005. ZINC- a free database of commercially available compounds for virtual screening. *Journal of chemical information and modeling* 45, 1 (2005), 177–182.

[25] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. 2022. VSGM: view-based GPU-accelerated subgraph matching on large graphs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[26] William L. Jorgensen, David S. Maxwell, and Julian Tirado-Rives. 1996. Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids. *Journal of the American Chemical Society* 118, 45 (1996), 11225–11236. doi:10.1021/ja9621760

[27] Jonas H. M"uller Kornd"orfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. 2022. LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 830–841. doi:10.1109/TPDS.2021.3107775

[28] Paul D. Leeson and Brian Springthorpe. 2007. The Influence of Drug-like Concepts on Decision-Making in Medicinal Chemistry. *Nature Reviews Drug Discovery* 6, 11 (2007), 881–890. doi:10.1038/nrd2445

[29] Matthew Leinhauser, René Widera, Sergei Bastrakov, Alexander Debus, Michael Bussmann, and Sunita Chandrasekaran. 2022. Metrics and Design of an Instruction Roofline Model for AMD GPUs. *ACM Trans. Parallel Comput.* 9, 1 (Jan. 2022). doi:10.1145/3505285

[30] Jiajie Li, Jan-Niklas Schmelzle, Yixiao Du, Simon Heumos, Andrea Guarracino, Giulia Guidi, Pjotr Prins, Erik Garrison, and Zhiru Zhang. 2024. Rapid GPU-Based Pangenome Graph Layout. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press. doi:10.1109/SC41406.2024.00035

[31] Candida Manelfi, Valerio Tazzari, Filippo Lunghini, Carmen Cerchia, Anna Fava, Alessandro Pedretti, Pieter F. W. Stouten, Giulio Vistoli, and Andrea Rosario Beccari. 2024. "DompeKeys": A Set of Novel Substructure-Based Descriptors for Efficient Chemical Space Mapping, Development and Structural Interpretation of Machine Learning Models, and Indexing of Large Databases. *Journal of Cheminformatics* 16, 1 (2024), 21. doi:10.1186/s13321-024-00813-4

[32] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2020. The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In *International Conference on Graph Transformation*. Springer, 316–324.

[33] Jeffrey Mendenhall, Benjamin P. Brown, Sandeepkumar Kothiwale, and Jens Meiler. 2021. BCL::Conf: Improved Open-Source Knowledge-Based Conformation Sampling Using the Crystallography Open Database. *Journal of Chemical Information and Modeling* 61, 1 (2021), 189–201. doi:10.1021/acs.jcim.0c01140

[34] NVIDIA. 2025. NVIDIA DCGM. https://developer.nvidia.com/dcgm.

[35] Noel M O'Boyle, Michael Banck, Craig A James, Chris Morley, Tim Vandermeersch, and Geoffrey R Hutchison. 2011. Open Babel: An open chemical toolbox. *Journal of cheminformatics* 3 (2011), 1–14.

[36] Linus Pauling. 1988. *General chemistry.* Courier Corporation.

[37] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019), 947–958.

[38] Jay W. Ponder and David A. Case. 2003. Force Fields for Protein Simulations. In *Advances in Protein Chemistry*. Vol. 66. Elsevier, 27–85. doi:10.1016/S0065-3233(03)66002-X

[39] A. K. Rappe, C. J. Casewit, K. S. Colwell, W. A. Goddard, and W. M. Skiff. 1992. UFF, a Full Periodic Table Force Field for Molecular Mechanics and Molecular Dynamics Simulations. *Journal of the American Chemical Society* 114, 25 (1992), 10024–10035. doi:10.1021/ja00051a040

[40] David Rogers and Mathew Hahn. 2010. Extended-connectivity fingerprints. *Journal of chemical information and modeling* 50, 5 (2010), 742–754.

[41] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. 2020. Self-supervised graph transformer on large-scale molecular data. *Advances in neural information processing systems* 33 (2020), 12559–12571.

[42] Robert Schmidt, Emanuel S. R. Ehmki, Farina Ohm, Hans-Christian Ehrlich, Andriy Mashychev, and Matthias Rarey. 2019. Comparing Molecular Patterns Using the Example of SMARTS: Theory and Algorithms. *Journal of Chemical Information and Modeling* 59, 6 (2019), 2560–2571. doi:10.1021/acs.jcim.9b00250

[43] Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. 2016. Load Balancing for Molecular Dynamics Simulations on Heterogeneous Architectures. In *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*. IEEE Computer Society, 101–110. doi:10.1109/HIPC.2016.021

[44] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[45] John C. Shelley, Anuradha Cholleti, Leah L. Frye, Jeremy R. Greenwood, Mathew R. Timlin, and Makoto Uchimaya. 2007. Epik: A Software Program for pK a Prediction and Protonation State Generation for Drug-like Molecules. *Journal of Computer-Aided Molecular Design* 21, 12 (2007), 681–691. doi:10.1007/s10822-007-9133-z

[46] Richard A Snay. 1976. Reducing the profile of sparse symmetric matrices. *Bulletin géodésique* 50, 4 (Dec. 1976), 341–352.

[47] Yunhao Sun, Guanyu Li, Jingjing Du, Bo Ning, and Heng Chen. 2022. A Subgraph Matching Algorithm Based on Subgraph Index for Knowledge Graph. *Frontiers of Computer Science* 16, 3 (2022), 163606. doi:10.1007/s11704-020-0360-y

[48] Benjamin I Tingle, Khanh G Tang, Mar Castanon, John J Gutierrez, Munkhzul Khurelbaatar, Chinzorig Dandarchuluun, Yurii S Moroz, and John J Irwin. 2023. ZINC-22—A free multi-billion-scale database of tangible compounds for ligand discovery. *Journal of chemical information and modeling* 63, 4 (2023), 1166–1176.

[49] Nishith Tirpankar and Hari Sundar. 2018. Towards Triangle Counting on GPU using Stable Radix binning. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. IEEE, 1–6. doi:10.1109/HPEC.2018.8547543

[50] TOP500.org. 2024. *November 2024 | TOP500.* https://top500.org/lists/top500/2024/11/

[51] Nenad Trinajstic. 2018. *Chemical graph theory.* CRC press.

[52] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[53] K. Vanommeslaeghe, E. Hatcher, C. Acharya, S. Kundu, S. Zhong, J. Shim, E. Darian, O. Guvench, P. Lopes, I. Vorobyov, and A. D. Mackerell. 2010. CHARMM General Force Field: A Force Field for Drug-like Molecules Compatible with the CHARMM All-atom Additive Biological Force Fields. *Journal of Computational Chemistry* 31, 4 (2010), 671–690. doi:10.1002/jcc.21367

[54] Junmei Wang, Romain M. Wolf, James W. Caldwell, Peter A. Kollman, and David A. Case. 2004. Development and Testing of a General Amber Force Field. *Journal of Computational Chemistry* 25, 9 (2004), 1157–1174. doi:10.1002/jcc.20035

[55] Yuyang Wang, Jianren Wang, Zhonglin Cao, and Amir Barati Farimani. 2022. Molecular contrastive learning of representations via graph neural networks. *Nature Machine Intelligence* 4, 3 (2022), 279–287.

[56] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[57] E.K. Wong. 1992. Model Matching in Robot Vision by Subgraph Isomorphism. *Pattern Recognition* 25, 3 (1992), 287–303. doi:10.1016/0031-3203(92)90111-U

[58] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. 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*. ACM, 1–14. doi:10.1145/3458817.3476214

[59] Xinxing Yang, Genke Yang, and Jian Chu. 2024. GraphCL-DTA: a graph contrastive learning with molecular semantics for drug-target binding affinity prediction. *IEEE Journal of Biomedical and Health Informatics* 28, 8 (2024), 4544–4552.

[60] Li Zeng, Lei Zou, and M Tamer Özsu. 2022. SGSI–A Scalable GPU-friendly Subgraph Isomorphism Algorithm. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2022), 11899–11916.

[61] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly subgraph isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1249–1260.