RUNTIME AND COMPILER OPTIMIZATIONS FOR SUBGRAPH MATCHING ALGORITHMS ON GPUS

by

Yihua Wei

A thesis submitted in partial fulfillment
of the requirements for the Doctor of Philosophy
degree in Computer Science in the
Graduate College of
The University of Iowa

May 2026

Thesis Committee: Peng Jiang, Thesis Supervisor
Bijaya Adhikari
Steve Goddard
Kasturi Varadarajan
Kasturi Varadarajan

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 MATCHA: A LANGUAGE AND COMPILER FOR BACKTRACKING-BASED SUBGRAPH MATCHING

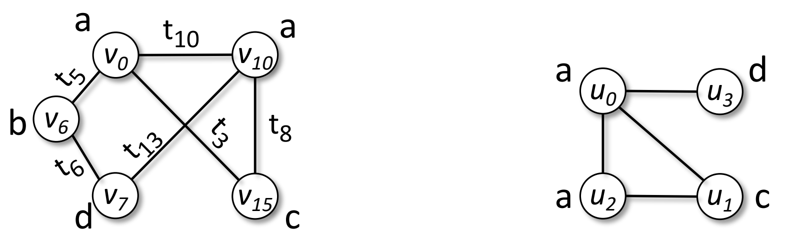
1.1 Introduction

Subgraph matching is the key building block of many graph analytics and learning applications. It aims to find subgraphs in a data graph conforming to the structural constraints of a query pattern. A subgraph matching algorithm can be used to extract information from graph-structured data in many domains, including bioinformatics [20], social network analysis [27], and cybersecurity [22]. Recently, subgraph matching also finds increasing use in graph-based machine learning applications, such as anomaly detection [2,21], entity resolution [4], and community detection [24].

Fig. 1 shows an example of the most basic subgraph matching problem. Given a data graph G and a query pattern Q, the task is to find all subgraphs in G that match the pattern Q. Both G and Q here have labeled vertices. There are two subgraphs in G with the same structure as Q. One subgraph $(v_{10,}v_{15}, v_0, v_7)$ is a valid match, while the other $(v_0, v_{10}, v_{15}, v_6)$ is not due to mismatched labels.

As an NP-hard problem, subgraph matching can be time-consuming on large graphs. Following the basic idea of backtracking [28], many advanced algorithms have been developed [1,6, 11,12,14]. Most of these algorithms are implemented on CPU, while some recent systems utilize GPU to accelerate computation [8,12,17,18,25,26].

Table 1 compares the state-of-the-art subgraph matching systems regarding their supported tasks, algorithms, and optimizations. Since each system features unique algorithms and optimizations, it has been increasingly difficult to compare and combine different solutions, thus hindering progress in this research area. For example, RapidMatch [26] proposes a relation-filtering technique with join plan optimization. Its join-based algorithm performs better than many other subgraph matching systems on CPU. However, migrating the algorithm to GPU is nontrivial. It is unclear whether the algorithm can outperform an efficient implementation of more basic algo-



(a) Data Graph G (b) Query Q

Figure 1. Example data graph and query pattern.

rithms on a GPU (e.g., G2Miner [10]).

Although some previous systems aim to be general-purpose [7, 9, 10], they hard-code algorithms for different tasks and support limited computation patterns. For example, G2Miner [10] assumes the subgraph is always extended vertex by vertex using set operations. Users cannot express the join-based algorithm of RapidMatch [26] with its programming interface. It also does not support backtracking with edge extension as adopted by Everest [32].

To overcome the limitations of existing systems, we propose a domain-specific language called Matcha for implementing subgraph matching algorithms. The key language constructs in Matcha include a `List` data type and an `apply` operation that applies a user-defined function to items in the `Lists`. Our design is based on the observation that the backtracking procedure in subgraph matching algorithms can be modeled as nested data stream processing. A Matcha program stores intermediate subgraphs in a `List` and applies a function to each subgraph to generate a new `List` of larger subgraphs until the desired pattern size is reached. By allowing programmers to specify different exploration strategies with the `apply` operator, Matcha supports a broader range of subgraph matching algorithms than previous systems.

We implement a domain-specific compiler that generates efficient C++/CUDA code based on Matcha programs. Unlike previous systems that hard-code the optimizations, we decouple the optimizations from algorithm specification by implementing optimizations as transformations on the IR. The advantage of such a design is that it allows us to compose and configure optimizations for different subgraph matching tasks, achieving better performance and portability than previous systems. Finally, the optimized IR is translated into C++/CUDA code with a code generator.

In summary, the paper makes the following contributions:

- We propose a DSL called Matcha for expressing various backtracking-based subgraph matching algorithms.
- We implement a compiler that translates Matcha programs into C++/CUDA code that runs on CPU and GPU.
- We implement a number of optimization passes including operator fusion, code motion, and work stealing in our compiler to improve the performance of generated code.
- Our system provides a unified framework for developing and comparing subgraph matching algorithms.

The experiments show that 1) Matcha can be used to easily reimplement state-of-the-art subgraph systems, and the reimplementations perform comparably to the original systems when the same optimizations are applied; 2) Additional optimizations can be easily incorporated into Matcha programs, and the optimized re-implementations outperform the original systems by 23.7x on average; 3) Matcha simplifies the comparison of algorithms originally implemented on different platforms. For example, we find that the join-based algorithm proposed in RapidMatch does not have an obvious performance advantage over the basic vertex-extension backtracking algorithm on the GPU; 4) Developing new algorithms using Matcha is convenient. As an example, we combine a decomposition-based algorithm with the join-based algorithm of RapidMatch, which results in better performance than either algorithm alone.

1.2 Background

1.2.1 Subgraph Matching Problems

The data graph in many real applications can be abstracted as a 4-tuple $G = (V, E, L, T)$, where V represents a set of vertices, $E = \{(v_i, v_j) | v_i, v_j \in V\}$ is a set of edges, L is a labeling function that assigns labels to the vertices, and T is a function that assigns (label or timing) information

Table 1. Comparison of state-of-the-art subgraph matching systems: G2Miner (GM) [10], STMatch (STM) [30], DecoMine (DM) [8], RapidMatch (RM) [26], Everest (EV) [32], and our system Matcha (Mt), in terms of support for algorithms, performance optimizations, and matching tasks.

		GM	**STM**	**DM**	**RM**	**EV**	**Mt**
Algorithms	Vertex Ext.	✓	✓	✓	✓	x	✓
	Edge Ext.	x	x	x	✓	✓	✓
	Pattern Decomp.	✓	x	✓	x	x	✓
	Hash Join	x	x	x	✓	x	✓
Optimizations	Multi-threading	x	x	✓	x	x	✓
	GPU	✓	✓	x	x	✓	✓
	Code Motion	✓	✓	✓	x	x	✓
	Work Stealing	x	✓	x	x	✓	✓
	Dyn. Scheduling	✓	x	✓	x	x	✓
	Thread Group	x	x	x	x	x	✓
Tasks	Edge-Induced	✓	✓	✓	✓	✓	✓
	Vertex-Induced	✓	✓	✓	x	✓	✓
	Labeled	✓	✓	✓	✓	✓	✓
	Temporal	x	x	x	x	✓	✓

to the edges. A graph $G' = (V', E', L', T')$ is a *subgraph* of $G = (V, E, L, T)$ if $V' \subseteq V$, $E' \subseteq E$, $L'(v) = L(v), \forall v \in V'$, and $T'(e) = T(e), \forall e \in E'$. A subgraph G' is *vertex-induced* if all the edges in E that connect the vertices in V' are included E'. For example, $(v_{10,}v_{15}, v_0, v_7)$ is a vertex-induced subgraph of G in Fig. 1(a). A connected subgraph G' is *edge-induced* if it is not vertex-induced. All subgraph matching problems are defined based on the concept of graph isomorphism:

Definition 1 (Graph Isomorphism)**.** Two graphs $G_a = (V_a, E_a, L_a, T_a)$ and $G_b = (V_b, E_b, L_b, T_b)$ are isomorphic if there is a bijective function $f : V_a \Rightarrow V_b$ such that $(v_i, v_j) \in E_a$ if and only if $(f(v_i), f(v_j)) \in E_b$.

Depending on the usage of the label and edge information, there are various subgraph matching tasks in real applications. Some of the most common ones are:

- *k-motif counting*, which counts the number of all size-k unlabeled subgraphs in a data graph G.
- *Subgraph listing*, which lists all the subgraphs in a data graph G that are isomorphic to a given query graph Q. The query can be either labeled or unlabeled. For labeled queries, the subgraphs must match the labels of vertices in Q.

- *k-clique listing*, which lists all the fully-connected unlabeled subgraphs of size-k in a graph G.
- *Temporal subgraph listing*, which lists all the subgraphs in a graph $G = (V_g, E_g, L_g, T_g)$ that match a temporal query $Q = (V_q, E_q, L_q)$. Given an ordering of connected edges in E_q (i.e., $E_q = \{e_i\}_{i=1}^{l}$), $T_g(f(e_i))$ must not exceed $T_g(f(e_{i+1}))$. A subgraph in G matches Q iff it is an isomorphism f with $T_g(f(e_l)) - T_g(f(e_1)) \leq \delta$, and $T_g(f(e_{i+1})) - T_g(f(e_i)) \leq \delta_i, \forall i \in [1, l)$, where δ is the time constraint set for Q, and δ_i is the time constraint set for each edge in Q.

1.2.2 Subgraph Matching Algorithms

We focus on backtracking-based algorithms for subgraph matching. While some ML-based algorithms have emerged in recent years [15, 16, 29], they are not as accurate as traditional combinatorial algorithms and are beyond the scope of this paper.

The combinatorial algorithms considered in this work follow a backtracking idea [28] – starting from an empty subgraph, the algorithm gradually extends the subgraph to the size of the query pattern. The most basic backtracking algorithm extends the subgraph by a vertex at each step. Take the query pattern in Fig. 1(b) as an example. The algorithm first finds all the vertices in the data graph that have the same label as u_0. If the query is unlabeled, it will be all vertices in the data graph. Next, starting from the size-1 subgraphs, the algorithm finds all vertices that match u_1. Since u_1 is connected to u_0, the matching vertices must be neighbors of the first vertex. This gives us a set of size-2 subgraphs that match (u_0, u_1). The algorithm continues to find vertices that match u_2. Since u_2 is connected to both u_0 and u_1, the vertices that match u_2 should be neighbors of both the first two matching vertices, and they can be computed with a set intersection operation. Last, the algorithm finds all matching vertices for u_3 based on its connectivity with u_0, u_1, u_2.

Based on the basic backtracking algorithm, many algorithm variants and optimizations have been proposed to improve the performance of specific subgraph matching tasks. For example, backtracking can be executed with edge extensions, where an intermediate subgraph is extended by an edge instead of vertex at each step. Such edge-extension algorithms are most naturally used for querying graph databases where the graph edges are stored as relations and the subgraph extension is achieved by joining the relations [1, 3, 14, 19]. More generally, an intermediate subgraph can be extended with another adjacent subgraph. This corresponds to algorithms that join two connecting subgraphs into a bigger subgraph [14, 19]. The pattern-decomposition algorithm [8, 23] can also be considered as extending a subgraph with one or more overlapping subgraphs. These algorithms also incorporate various optimizations to prune the subgraph exploration space as much as possible [5, 11, 12].

1.3 Limitations With Existing Subgraph Matching Systems

While numerous systems for subgraph matching have been proposed [1, 8, 17, 18, 23, 26, 30, 32], each of these systems features unique algorithms and optimizations. This makes it challenging to compare and combine existing techniques, and to develop new techniques based on prior work.

Restricted Algorithm Support. Previous subgraph matching systems implement algorithms that are best suited for specific tasks. For instance, DecoMine [8] employs a pattern-decomposition-based algorithm [23]; it is most efficient for counting subgraphs that can be divided into independent sub-patterns. RapidMatch [26] implements a join-based algorithm; it works best for labeled queries. AutoMine [18] supports both labeled and unlabeled queries, but it only runs the basic backtracking algorithm that matches the pattern vertices one by one. We summarize the algorithm support of state-of-the-art subgraph matching systems in Table 1.

Inconsistent Optimizations. The existing subgraph matching systems are inconsistent in performance optimizations. First, they are implemented on different platforms. Most of the systems are CPU-based [8, 12, 17, 18, 25, 26], while some recent systems utilize accelerators such as GPUs to speed up computation [10, 30, 31, 33]. This variety makes it difficult to compare and combine optimization techniques across different platforms. For instance, RapidMatch [26] is a CPU-only system. It is unclear whether its join-based algorithm can run efficiently on GPU and whether it can outperform the optimized GPU systems such as STMatch [30] and G2Miner [10] that use more basic vertex-extension algorithms. Even on the same hardware, different systems feature different optimizations, as summarized in Table 1.

Given the limitations of the existing systems, a unified framework that allows the development and integration of subgraph matching algorithms and optimization techniques across different platforms is desirable.

1.4 The Matcha Dsl

We propose a domain-specific language (DSL) called Matcha to express backtracking-based subgraph matching algorithms. The language supports four basic data types: `Var`, `Tuple`, `List`, and `Graph`. A `Var` represents a scalar variable, which can be an integer, floating-point, or boolean. A `Tuple` comprises one or multiple `Var`s of the same data type. A `List` is an iterable set of `Var`s or `Tuple`s. A `Graph` is a list of edges with additional information associated with vertices and/or edges. Matcha provides an API that allows users to perform various operations on the four data types. In this section, we describe the API using a Python-like syntax.

1.4.1 Programming Interface

Matcha supports basic arithmetic $(+,-,*,/)$ and comparison operations on integer and floating-point `Var`s, as well as logical operations (and, or, not) on boolean `Var`s. To create a `Tuple`, users specify the `Var`s in it with the **mk_tuple**$(nvars, dtype, val)$ operator. Here, *nvars* is the size of the `Tuple`, *dtype* is the data type of the `Var`s, and *val* is an optional argument for initializing the `Tuple`. The user can obtain a `Var` from a specific position in a `Tuple` t with the **get**(t, pos) operator.

Users can create a `List` in Matcha with the operator: **mk_list**$(nitems, type, val)$. Here, the *nitems* is an integer indicating the maximum number of items that can be stored in the `List`. The *type* argument specifies the type of the items, which can be `Var` or `Tuple`. The *val* argument is for initializing the `List`.

To define a `Graph`, users need to specify its edge list with the **mk_graph**$(edges, label, einfo)$ operator. Here, the *edges* must be a `List` of two-integer `Tuple`s representing the edges' source and destination. Optionally, the user can specify vertex and edge information on the graph. The *vlabel* argument is also a `List` of two-integer `Tuples`, where the first number is the vertex ID and the second number is the label of that vertex. The argument *einfo* is a `List` of the same length as *edges*; it associates a `Var` to each edge in the edge list.

Matcha supports a variety of operations on `List` and `Graph`, which are essential for implementing subgraph matching algorithms:

- **neighbor**(*g*: `Graph`, *v*: `Var<int>`). This operator returns the neighboring vertices of a vertex *v* in a Graph *g*. The returned neighbors are stored as a `List<Var<int>>`.
- **edge_info**(g: Graph, v: `Var<int>`) returns the information on the neighboring edges of a vertex *v* as a `List<Var>`.
- **vertex_label**(*g*: Graph, *v*: `Var<int>`) returns the label of a vertex *v* as a `Var<int>`.
- **apply**(*func*: `Func(T1,...)`$\rightarrow$`Tout`, *cond*: `Func(T1,...)` $\rightarrow$`Var<bool>`, *l1*: `List<T1>`, ...)$\rightarrow$`List<Tout>`. This is the most important operator in Matcha. It accepts one or multiple `Lists` as the input and iterates over items in all the `Lists` at the same time. It performs computation on each item with a function *func*, and returns a new `List` that contains the computed results. If the *cond* function is provided, the operator evaluates the *cond* function first and only computes *func* on items where the condition is `True`.
- **intersect**(*l1*: `List<Var<T>>`, *l2*: `List<Var<T>>`) $\rightarrow$ `List<Var<T>>` computes the intersection of two `Lists` of `Vars`. An item *t* is in the resulting `List` if and only if it is in both $l1$ and $l2$. This operator is commonly used in subgraph matching algorithms to obtain nodes connected to two previously matched nodes.
- **diff**(*l1*: `List<Var<T>>`, *l2*: `List<Var<T>>`) $\rightarrow$ `List`

  `<Var<T>>`. Similar to `intersect`, this operator processes two `Lists` of `Vars`, but it computes the difference between the two `Lists`. An item *t* is in the result if it is in $l1$ but not in $l2$.
- **sum**(*l*: `List<Var<T>>`) $\rightarrow$ `Var<T>` sums up the `Var` items in a `List`. The operator is convenient for subgraph counting tasks.

```
1  def RapidMatch(L01, L02, L03, L12):
2    G02 = mk_graph(edges = L02)
3    G03 = mk_graph(edges = L03)
4    G12 = mk_graph(edges = L12)
5    def f1(e):
6      S0 = neighbor(G02, get(e, 0))
7      S1 = neighbor(G12, get(e, 1))
8      C2 = intersect(S0, S1)
9      def f2(v2):
10       C3 = diff(neighbor(G03, get(e, 0)), mk_list(val=[get(e, 1), v2]))
11       return size(C3)
12     return sum(apply(func=f2, cond=None, C2))
13   return sum(apply(func=f1, cond=None, L01))
```

Listing 1. Join-based algorithm implemented in Matcha for the query in Fig. 1(b).

- **size**(l: `List|Tuple`) $\rightarrow$ `Var<int>` returns the number of items in a `List` or `Tuple`.

Examples: Listing 1 shows a Matcha implementation of a join-based algorithm [26] for the query in Fig. 1(b). The algorithm takes as input the filtered edge `Lists` ($L01$, $L02$, $L03$, $L12$) that match the four edges of the query pattern. The `mk_graph` operations in line 2-4 build the index data structures for the edge lists corresponding to (u_0, u_2), (u_0, u_3) and (u_1, u_2). We will explain how the index data structure is constructed in Section 1.5.1. In line 13, the algorithm starts by iterating over the edges in $L01$. For each edge, it computes the set of vertices $C2$ that match u_2 by intersecting neighbors of v_0 and v_1 (line 8). Next, for each vertex in $C2$, it counts the vertices that match u_3 (line 9-11). The counts are summed up to obtain the final result.

Listing 2 shows another example of Matcha implementing a decomposition-based algorithm [8, 23] for the query in Fig 1(b). Since the algorithm takes the entire graph as input, it needs to first filter out the edges that do not match the labels of (u_0, u_1). This can be achieved by an `apply` operator with a filtering condition (line 15). For each matching edge, it computes the vertices that match u_2 by intersecting neighbors of v_0 and v_1 and filtering out the vertices with labels different from u_2 (line 10). The vertices that match u_3 are the neighbors of v_0 with the label of u_3 (line 11). Next, since u_2 and u_3 are isolated by (u_0, u_1) in the query pattern, we can directly calculate the number of matching subgraphs on an edge as `size(C2)` $*$ `size(S3)` (line 12). The counts from different edges are summed to obtain the final result.

```
def Decomine(edges):
  G = mk_graph(edges)

  # select vertices of a certain label from S
  def select(S, label)
    return apply(lambda v: v, cond=lambda v: vertex_label(G, v) == label,
       S)
  def f(e):
    S0 = neighbor(G, get(e, 0))
    S1 = neighbor(G, get(e, 1))
    C2 = select(intersect(S0, S1), 'a')
    S3 = select(S0, 'd')
    return size(C2) * size(S3)

  # iterate over edges that match (u0, u1)
  return sum(apply(func=f, cond=lambda e: vertex_label(G, get(e, 0)) ==
     'a' and vertex_label(G, get(e, 1)) == 'c', edges))
```

Listing 2. Decomposition-based algorithm implemented in Matcha for the query in Fig. 1(b).

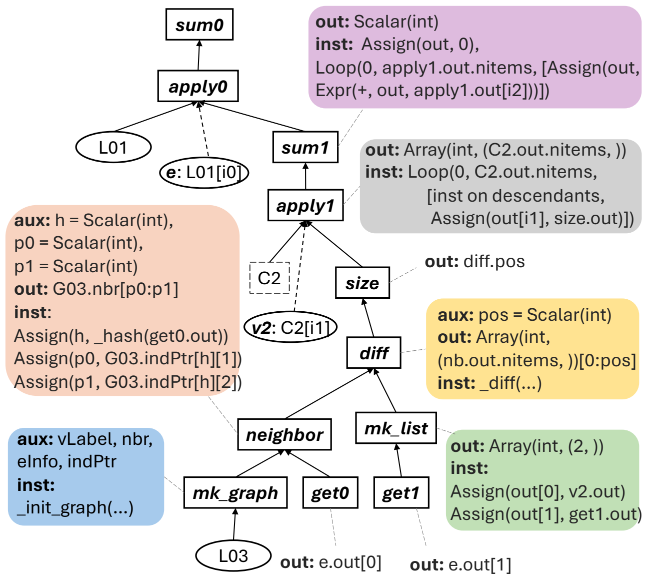
1.5 Matcha Compilation Pipeline

A Matcha program can be depicted as an Abstract Syntax Graph (ASG). In this graph, nodes represent data objects and operators, while edges connect each operator to its inputs. The Matcha compiler generates an intermediate representation (IR) from the ASG. The IR may go through several optimization stages before converting into C++/CUDA code.

Fig. 2 shows an example of the compilation pipeline. The program in Listing 1 is depicted as an ASG in Fig. 2(a). The `apply` operator on the top (`apply0`) has $f1$ as the computation function and $L01$ as the input `List`. The items in $L01$ are referenced by $f1$ through an operand e attached to the `apply` operator. Within the subtree of the function $f1$, the inner apply operator (`apply1`) has $f2$ as the computation function and $C2$ as the input `List`. The function $f2$ accesses items in $C2$ through an operand $v2$ attached to `apply1`. For a simple illustration, we omit the subtree for computing $C2$ in the figure.

1.5.1 Generating Ir On Asg

The compiler traverses the ASG in a depth-first order to generate the IR on each node. This ensures that the IR on any node is generated after the IR on all its input nodes has been generated.



(a) ASG to IR

```
int RapidMatch(..., int L03[][2],
    int len3, ...) {
 // Memory Pre-allocation
 auto G03_nbr = new int[len3];
 auto G03_indPtr = new int[H][3];
 int nb_h, nb_p0, nb_p1;
 auto lst = new int[2];
 int df_pos;
 auto df_out = new int[_max_degree(L03)];
 auto apl1_out = new int[_max_degree(L01)];
 int sum1;
 ...
 for (int i0 = 0; i0 < len1; i0++) {
  ...
  for (int i1 = 0; i1 < C2_size; i1++) {
   _init_graph(L03, len3, G03_nbr,
               G03_indPtr);
   nb_h = _hash(L01[i0][0]);
   nb_p0 = G03_indPtr[nb_h][1];
   nb_p1 = G03_indPtr[nb_h][2];
   lst[0] = C2[i1];
   lst[1] = L01[i0][1];
   _diff(df_out, df_pos, G03_nbr,
         nb_p0, nb_p1, lst);
   apl1_out[i1] = df_pos; }
  sum1 = 0;
  for (int i2 = 0; i2 < C2_size; i2++) {
   sum1 = sum1 + apl1_out[i2];} }...}
```

(b) Generated C++ code

Figure 2. Compiling a Matcha program to C++ code. The IR of each ASG node contains an output (out), some auxiliary data (aux), and instructions (inst) for computing the output.

Upon visiting a node, the compiler determines the output of the operator as well as instructions for computing the output.

The IR has two data types: `Scalar` and `Array`. A `Scalar` can be integer, floating-point, or boolean. An `Array` can be one-dimensional or multi-dimensional and is defined by its data type and the size of each dimension. Each element in an `Array` is a `Scalar` and can be accessed through array `Indexing`. To represent arithmetic, comparison, and logical expressions, the IR has an `Expr(op, lhs, rhs)` construct where the *op* is the operation type, and the *lhs*/*rhs* can be a `Scalar`, an `Expr`, or a constant literal. The IR also has an `Assign(dest, src)` construct for representing assignments. Due to space limit, we only describe the generation rules for the `Graph` operators in this section. The rules for other operators are straightforward as illustrated in Fig. 2.

When creating a `Graph` object, we construct an auxiliary data structure based on the input *edges*, *vlabel*, and *einfo*. The data structure helps achieve efficient access to neighboring information in the graph. Specifically, upon visiting a `mk_graph` node on the ASG, the compiler generates

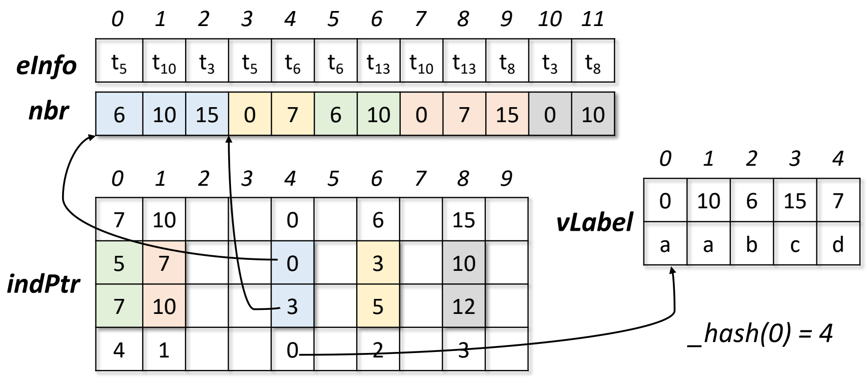


Figure 3. Graph G from Fig. 1(a) stored in four `Arrays`.

four auxiliary `Arrays` ($vLabel$, nbr, $eInfo$, $indPtr$). The $vLabel$ is a copy of the input $vlabel$ (if the information is provided); it is a 2-D array that pairs unique vertex IDs with their labels. The nbr is a 1-D array that holds the neighbors of vertices corresponding to the vertex IDs in $vLabel$. The neighbors of each vertex are stored contiguously in nbr. The $eInfo$ Array has the same size as nbr, and it contains information on the neighboring edges corresponding to vertices in nbr. Fig. 3 shows an example of the data structure for the graph G in Fig. 1(a).

Depending on whether the vertex IDs are dense, we build different indirection arrays to achieve an efficient lookup of vertices' neighbors. If the vertex IDs are contiguous natural numbers, we use a 1-D Array ($indPtr$) to store the boundaries of their neighbors in nbr. The starting position of v's neighbors in nbr is $indPtr[v]$. This data structure is exactly the Compressed Sparse Row (CSR) format commonly used for storing graph data.

However, for edges stored as relational tables, which are commonly used in join-based algorithms [26], the vertex IDs are non-contiguous and sparse. To efficiently locate neighbors for the vertices, we store the neighbor boundaries in a hash table. Specifically, the $indPtr$ is a 2-D Array of size $(H \times 4)$ where H is the number of slots in the hash table. For a vertex v, its neighbor boundaries are stored in $indPtr[\_hash[v]][1]$ and $indPtr[\_hash[v]][2]$, and its position in $vLabel$ is stored in $indPtr[\_hash[v]][3]$. In Fig. 3, the neighbors of vertex-0 are stored from position-0 ($indPtr[\_hash(0)][1]$) to position-3 ($indPtr[\_hash(0)][2]$) in nbr. The initialization of the data structure is performed by a built-in function ($\_init\_graph$), which is invoked on the `mk_graph` node.

The `neighbor`, `edge_info`, and `vertex_label` operators can be implemented efficiently with *indPtr*. The output of `neighbor/edge_info` is a slice of the graph's *nbr*/*eInfo* Array. The output of `vertex_label` operator is a `Scalar` read from *vLabel*. If the graph is stored in CSR format, the slicing boundaries for vertex v are $indPtr[v]$ and $indPtr[v+1]$; otherwise, the slicing boundaries are obtained from the hash table.

1.5.2 Translating Ir To C++ Code

The IR needs to undergo several optimization passes before being translated to the target code. We leave the details of code optimizations/parallelization in the next section and focus on generating sequential CPU code in this section.

After the IR is generated, the compiler takes another depth-first traversal on the ASG and generates a C++ function based on the IR. The leaf nodes (i.e., nodes without any input) are input data (e.g., $L01$, $L03$ in Fig. 2(a)). These data are initialized by the host code and are passed as arguments to the C++ function. In our implementation, the input data are initialized in Python, and the generated C++ code is compiled just-in-time into a pybind11 [13] module, which is invoked by the Python program.

Memory Pre-allocation. If an internal node on the ASG contains an `Array`, we need to allocate memory for the data. A straightforward approach is to allocate memory on the node. However, this involves dynamic memory allocation and may incur a large overhead, especially when it is in a loop. For example, the `intersect` operator in Listing 1 needs to allocate memory for the intersection results every time $f1$ is executed (i.e., for every item in $L01$). To avoid repeated memory allocation, we pre-allocate memory for `Arrays` on all ASG nodes at the beginning of the generated code. Specifically, the compiler checks the size of every `Array` in the internal nodes. If the size is constant or can be determined based on the data in the leaf nodes, we can pre-allocate the memory before execution of the internal nodes. If the size is unknown, the compiler reports an error and asks the user to provide a size.

Once memory is allocated, generating computing instructions is straightforward. The `Expr`, `Loop`, and `Assign` constructs in the IR can be directly translated to expressions, for-loops, and assignments in C++. If a node is not a descendant of an `apply` operator, its *inst* is immediately translated when the node is visited. However, if it is a descendant of an `apply` operator, the node is skipped since its instructions should be included in the loop body of the `apply` operator. Fig. 2(b) shows the generated C++ code corresponding to the IR in Fig. 2(a).

1.6 Optimization And Parallelization

Matcha decouples the definition of subgraph matching algorithm from its execution strategy. This section explains how various optimization and parallelization strategies can be applied automatically to Matcha programs through IR transformations.

1.6.1 Operator Fusion

Our compiler fuses four types of operator pairs:

- `apply+sum`. Consider the loop of `apply1` (colored in grey) and the loop of `sum1` (colored in purple) in Fig. 2(b). The two loops share the same iteration space. The `apply1` loop writes data to $apl1\_out$, which is read by the `sum1` loop. We can fuse the two loops to avoid the storage of $apl1\_out$.

- `intersect/diff+sum`. In Fig. 2(a), we can also see that the output array of `diff` operator is never used; the algorithm is only interested in the number of items in the array. In this case, we can remove the allocation of df_out and invoke a more efficient version of $\_diff$ without storing the output data.

- `neighbor+apply` with boundary conditions. This pair of operators often appear in labeled subgraph matching algorithms, as they need to select neighboring vertices with a specific label (e.g., line 11 in Listing 2). By fusing the two operators, we can avoid the allocation of a new `Array` for the output of `apply` and replace it with a slice of the neighbor list.

- `intersect/diff+apply` with boundary conditions. An `intersect/diff` operator can also be fused with a conditional `apply` if the condition only affects the slicing boundaries of the output. This fusion enables the efficient implementation of the symmetry-breaking technique that eliminates redundant subgraphs [17].

The fusion is applied to the ASG nodes in a depth-first order. For each pair of child-parent nodes, we check if they match any fusible types and perform fusion on the IR accordingly.

1.6.2 Loop-invariant Code Motion

Consider the *_init_graph* function in the `apply1` loop in Fig. 2(b). The function's execution does not depend on the loop iterate $i1$, which means that it can be moved outside of the loop. In fact, the function does not depend on the loop iterate $i0$ either and can be moved outside of the `apply0` loop. This optimization is often referred to as loop-invariant code motion in compiler optimization literature.

Besides code motion at the ASG level, there might also be loop-invariant instructions at the IR level. For example, the `mk_list` node in Fig. 2(a) contains two `Assign`s, and the second `Assign` does not reference the items of $C2$. In such cases, we only include the first `Assign` in the loop body of `apply1` and leave the second `Assign` within the `mk_list`.

This optimization also achieves the code motion of set operations proposed in previous work [18]. For any `intersect` or `diff` operators independent from the input of the `apply` operator, the compiler automatically moves the code outside the loop. Listing 3 shows the optimized code after operator fusion and code motion for the example in Fig. 2.

1.6.3 Automatic Parallelization

Following previous work [10, 30], we apply two-level parallelization to the subgraph exploration procedure. The first parallelization is applied to the outermost loop that iterates over the initial vertices/edges. On a CPU, this corresponds to using multiple threads to explore subgraphs from different vertices/edges simultaneously. The second parallelization is applied to the compu-

```
int RapidMatch(..., int L03[][2]) {
 ...
 int sum1;
 ...
 //_init_graph is moved out of i0 loop after code motion.
 _init_graph(L03, len3, G03_nbr, G03_indPtr);
 for (int i0 = 0; i0 < len1; i0++) {
    ...
    //The four statements are moved out of i1 loop after code motion.
    nb_h = _hash(L01[i0][0]);
    nb_p0 = G03_indPtr[nb_h][1];
    nb_p1 = G03_indPtr[nb_h][2];
    lst[1] = L01[i0][1];
    sum1 = 0;
    //i1 loop is fused into i2 loop.
    for (int i2 = 0; i2 < C2_size; i2++) {
        lst[0] = C2[i2];
        _diff(df_out, df_pos, G03_nbr,
        nb_p0, nb_p1, lst);
        sum1 = sum1 + df_pos; }
    apl0_out[i0] = sum1; } }
```

Listing 3. Optimized code after operator fusion and code motion for the example in Fig 2.

tation of intermediate subgraphs at each exploration step. It exploits fine-grained parallelism (i.e., SIMD) to accelerate the `intersect/diff` operations.

GPU parallelization is similar. Unlike previous work that uses either a warp [10, 30] or a single thread [32] as the execution unit, we use cooperative thread groups [**?**] on Nvidia GPU to execute iterations of the outermost loop in parallel. This implementation allows us to configure the thread-group size and achieve better GPU utilization. For `intersect/diff`, we adapt the code from G2Miner [10] to use threads in each group for parallel execution.

Work Stealing. Subgraph matching on a GPU usually suffers from severe load imbalance [30]. Work-stealing is an effective technique that can mitigate this issue and significantly improve performance. We implement the work-stealing strategy from [30] in our compiler. Specifically, we place the *aux* in the IR of `neighbor`, `intersect` and `diff` operators, as well as the iterators of all `apply` loops in GPU shared memory. This allows an idle thread-group to check the amount of remaining work in another thread-group and steal work from it.

1.7 Evaluation

1.7.1 Experimental Setup

Platform. Our experiments are conducted on a dual-socket machine with Intel Xeon Gold 6226R 2.9GHz CPUs (32 cores in total), 512GB RAM, and an Nvidia A100 GPU. The generated CPU code was compiled using GCC 9.4.0, and the generated GPU code was compiled using NVCC 11.2.

Subgraph Matching Tasks. We evaluate the performance of four representative subgraph matching tasks: k-motif counting (k-MC), subgraph listing (SL), k-clique counting (k-CL), and temporal subgraph listing (TSL). A more detailed description of the tasks can be found in Section 1.2.1.

Datasets. Table 2 lists the data graphs used in our experiments. Six static graphs are used for k-MC, SL, and k-CL, and three temporal graphs are used for TSL. These graphs are commonly used to evaluate subgraph matching systems in previous work [7, 8]. For labeled SL, we randomly assign five labels to the vertices in the graphs.

Table 2. Graph datasets.

Graph	**# nodes**	**# edges**	**Max degree**
MiCo (mc)	96K	1.1M	1359
DBLP (db)	317K	1.0M	343
Amazon (az)	334K	0.9M	549
YouTube (yo)	1.1M	3.0M	28754
Patent (pt)	3.8M	16.5M	793
LiveJournal (lj)	4.0M	34.7M	14815
EmailEuCore (ee)	1K	332K	9782
wikitalk (wt)	1.1M	7.8M	264905
StackOverflow (st)	2.6M	63.5M	101663

Compared Systems. We compare our system against five state-of-the-art systems: STMatch (**STM**) [30], RapidMatch (**RM**) [26], DecoMine (**DM**) [8], G2Miner (**GM**) [10], and Everest (**EV**) [32]. These systems were chosen for their diverse algorithm and optimization supports. GM and STM implement the vertex-extension backtracking algorithm for general subgraph listing and motif counting tasks. DM employs a pattern-decomposition-based algorithm for subgraph counting. EV implements an edge-extension backtracking algorithm for temporal subgraph listing. RM

Table 3. Execution time of motif counting with vertex-extension algorithm. ‘ms’, ‘s’, and ‘m’ stand for milliseconds, seconds, and minutes.

		mc	**db**	**az**	**yo**	**pt**	**lj**
3-MC	**GM**	2.6ms	1.2ms	1.0ms	18ms	23ms	142ms
	mGM	2.7ms	1.2ms	1.1ms	18ms	23ms	141ms
	mGM-o	1.7ms	0.6ms	0.4ms	13ms	16ms	136ms
4-MC	**GM**	1.58s	0.11s	0.04s	227s	1.93s	415s
	mGM	1.57s	0.13s	0.05s	227s	1.93s	415s
	mGM-o	1.02s	0.06s	0.02s	192s	1.78s	407s

Table 4. Execution time of motif counting with a decomposition-based algorithm.

		mc	**db**	**az**	**yo**	**pt**	**lj**
4-MC	**mDM-cpu**	0.39s	64ms	43ms	0.74s	1.6s	33s
	mDM-gpu	0.03s	3ms	2ms	0.06s	0.09s	2.3s
5-MC	**mDM-cpu**	128s	1.3s	0.17s	576s	21s	212m
	mDM-gpu	22s	0.3s	0.04s	17s	3.3s	26m

employs a join-based algorithm for labeled subgraph listing. GM, STM, and EV are GPU-based systems, while DM and RM only run on CPU.

To demonstrate Matcha’s flexibility and efficiency, we reimplement the five systems in Matcha and compare performance with their original implementations. These reimplementations (referred to as **mGM**, **mDM**, **mSTM**, **mRM**, and **mEV**) use the same optimizations as original systems. Then, we improve the performance of the reimplementations by enabling various optimizations in our compiler. For the two CPU-only systems (DM and RM), we also parallelize the code for GPU execution.

1.7.2 Results for Motif Counting

We use DC and GM for motif counting as they reportedly achieve the best performance among existing systems for the task on CPU and GPU, respectively.

Table 3 lists the execution time of 3-MC and 4-MC with the vertex-extension algorithm of GM running on GPU. The results show that our reimplementation achieves almost the same performance as the original GM system, validating the efficiency of our generated code. We further optimize the baseline reimplementation by using a group of 8 threads to execute each iteration of

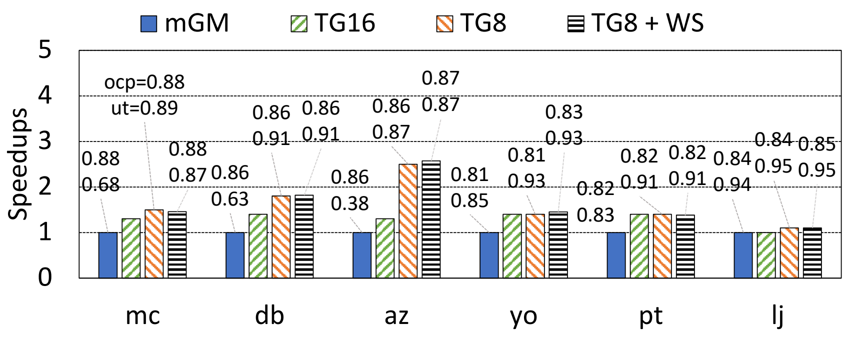


Figure 4. Effect of thread-grouping (TG) and work-stealing (WS) on mGM for 3-MC. ‘ocp’, ‘ut’ stand for SM occupancy and thread utilization.

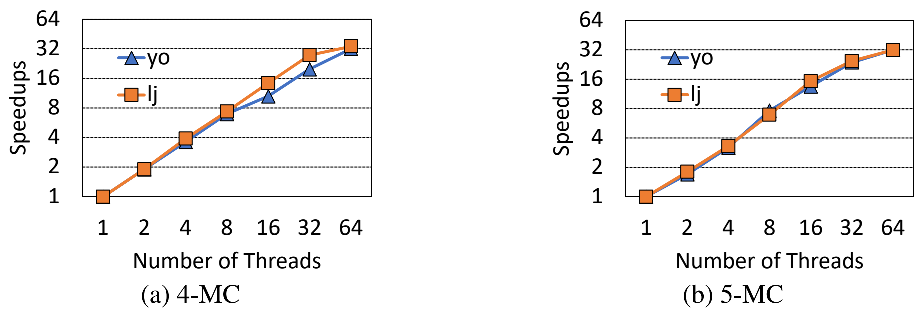


Figure 5. Scalability of multi-threaded mDM on CPU.

the outermost loop. We also enable work-stealing among the thread groups. (The original GM system uses warps for parallel execution without any work-stealing.) With the optimizations, our generated code (mGM-o) achieves 1.1x to 2.6x speedups (with an average of 1.5x) against the original GM.

Table 4 lists the execution time of 4-MC and 5-MC with the pattern-decomposition algorithm of DM. Since DM is not open-sourced, we cannot directly compare performance with their original implementation. The execution time of our re-implementation on CPU is close to the numbers reported in their paper, affirming the efficiency of our generated code. While the original DM is CPU-only, our system automatically generates GPU code for the algorithm, easily accelerating the execution by 4x to 53x (with an average of 27x).

By activating different optimization passes during Matcha compilation, we study the effectiveness of each optimization technique. Fig. 4 shows how the performance of mGM is affected by thread-grouping and work-stealing. We profile the execution with Nvidia NSight [**?**] and mark the SM occupancy and thread utilization data in the figure. Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported. Thread utilization is defined as the ratio of active threads in a warp to the total number of threads in the warp. As we can see from the figure, thread-grouping effectively improves thread utilization, bringing 1.1x to 2.5x speedups against the baseline mGM. On the other hand, work-stealing has little effect on thread utilization and SM occupancy. This is because the nested loop has only two levels for 3-MC, and the baseline mGM already achieves good load balance with dynamic scheduling of the outermost loop.

For DM, we study how the algorithm scales on a CPU with different numbers of threads. Fig. 5 shows that our generated code achieves near-linear speedups up to 32 threads. When the thread count increases to 64, performance still improves due to the CPU's hyper-threading.

1.7.3 Results for Subgraph Listing

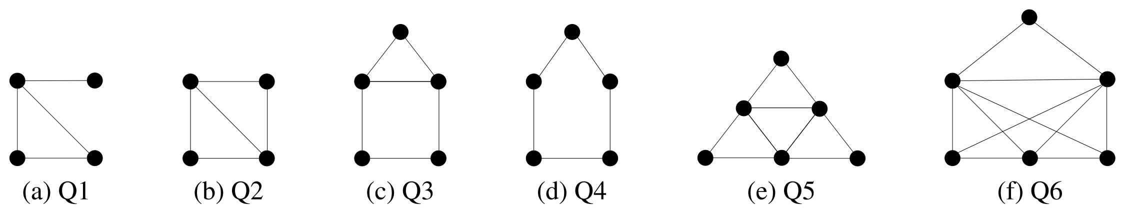


Figure 6. Query patterns for subgraph listing.

GM, STM, and RM all support general subgraph listing. We test and compare their performance with our Matcha implementation using the query patterns in Fig. 6.

Performance of Unlabeled Queries. Table 5 shows the execution time of different systems for listing unlabeled subgraphs of the query patterns. In all test cases, our re-implementation of STM (mSTM) either matches or surpasses the performance of the original STM system. This is because the original STM needs to maintain a stack data structure to simulate the recursive execution of the backtracking algorithm, while Matcha directly translates the algorithm into a nested loop. The performance advantage of mSTM diminishes with large query patterns because the overhead of maintaining the stack data structure becomes small compared to the computation itself.

GM performs especially well on small queries but is slower than STM for larger queries. This is because GM launches many more thread-blocks (7000+) than STM (82) and achieves

Table 5. Execution time of unlabeled subgraph listing with vertex extension. ‘s’ stands for seconds. All other numbers are in milliseconds. ‘–’ indicates timeout after 4 hours.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**STM**	12	14	71	80	138	132
	mSTM	10	5.4	71	16	131	131
	GM	2.6	2.7	41	8.4	132	164
	mGM	2.5	2.7	41	8.4	132	164
	mGM-o	1.3	1.0	18	3.7	131	76
db	**STM**	312	147	903	368	14.6s	29.4s
	mSTM	110	39	901	367	14.6s	28.7s
	GM	26	12	1220	388	67.0s	95.6s
	mGM	26	14	1232	386	67.0s	95.6s
	mGM-o	15	5.3	457	138	12.7s	20.5s
mc	**STM**	445	235	122s	30.3s	–	–
	M-STM	449	229	121s	30.0s	–	–
	GM	769	315	178s	35.6s	–	–
	mGM	774	317	178s	36.6s	–	–
	mGM-o	407	227	121s	29.3s	–	–

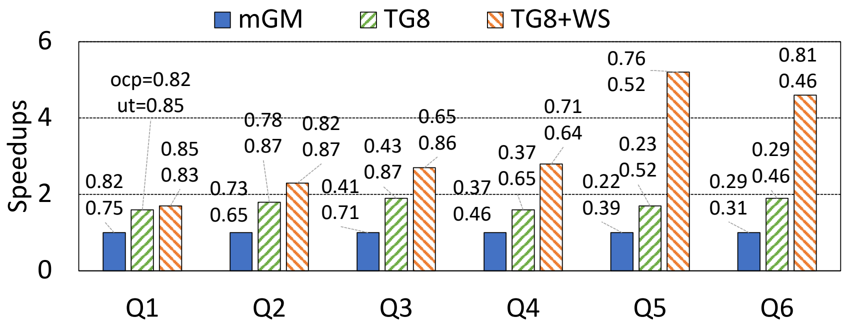


Figure 7. Effect of thread-grouping (TG) and work-stealing (WS) on mGM for subgraph listing on DBLP graph.

better GPU occupancy for small tasks, while STM achieves better load balance for larger tasks due to its work-stealing technique. Our re-implementation of GM (mGM) achieves almost the same performance as the original GM.

To further improve the performance, we apply thread-grouping and work-stealing to mGM. The optimized code (mGM-o) brings the best of both STM and GM, achieving up to 7.7x speedups against the original STM and 4.6x speedups against GM. Fig. 7 shows the effectiveness of each optimization. As expected, thread-grouping increases thread utilization, leading to 1.5x to 2.8x speedups against mGM. The benefit of work-stealing is small for the size-4 queries (Q1 and Q2), which is similar to the results for 3-MC in Fig. 4. However, as the pattern size grows (Q3∼Q6), the

Table 6. Execution time (in milliseconds) of labeled subgraph listing with vertex extension.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**STM**	12	12	12	10	9.5	6.9
	mGM-o	0.4	0.4	0.9	3.3	1.0	0.6
db	**STM**	12	12	14	19	61	83
	mGM-o	1.4	1.5	7.9	8.2	59	58
mc	**STM**	8.5	8.4	366	612	1371	1197
	mGM-o	5.3	5.1	272	383	1277	1183

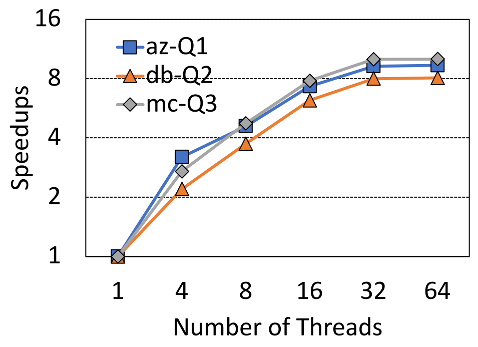
Table 7. Execution time of labeled subgraph listing with a join-based algorithm. 's' stands for seconds. All other numbers are in milliseconds.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**RM**	60	67	75	95	90	96
	mRM	12	14	12	18	18	17
	mRMo-cpu	1.3	1.6	1.6	2.9	1.6	1.6
	mRMo-gpu	0.1	0.1	0.3	1.2	0.2	0.2
db	**RM**	71	78	205	360	2262	1863
	mRM	13	17	66	130	971	1388
	mRMo-cpu	1.8	1.0	14	16	260	292
	mRMo-gpu	0.6	0.5	7.8	8.3	56	59
mc	**RM**	165	223	9688	18.1s	257s	207s
	mRM	62	136	7983	16.7s	239s	198s
	mRMo-cpu	8.5	18	797	1.1s	28s	28s
	mRMo-gpu	5.4	4.8	267	0.33s	1.25s	1.10s

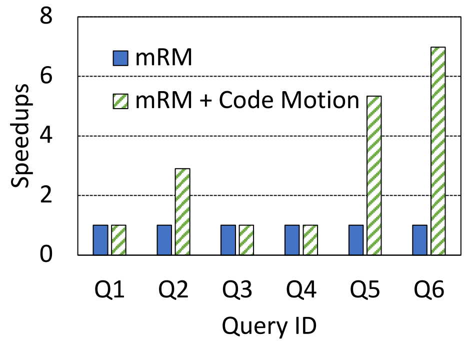
benefit becomes more noticeable. This performance is reflected by the SM occupancy. For Q1 and Q2, occupancy is already high without work-stealing. But for Q3∼Q6, work-stealing significantly increases occupancy.

Performance of Labeled Queries. Table 6 shows the performance of the backtracking algorithm for labeled subgraph listing. The queries are generated by randomly assigning four labels to the pattern vertices. Compared to STM, our code (mGM-o) is consistently faster for all queries, achieving an average speedup of 7.1x.

We also test the performance of the join-based algorithm of RM for the same set of labeled queries. The results are listed in Table 7. The original RM is a single-threaded CPU implementation. We reimplement it in Matcha and apply two optimizations, code motion and multi-threading, to it. The generated CPU code (mRMo-cpu) is on average 29.4x (up to 60x) faster than the original RM. Fig. 8 shows how each of the two optimizations affects the performance. For Q2, Q5, and Q6,



(a) Multi-threading



(b) Code motion on DBLP graph

Figure 8. Effect of code motion and multi-threading on RM for labeled subgraph listing.

Table 8. Execution time (in milliseconds) of labeled subgraph counting with decomposition-based and join-based algorithms.

Graph		Q1	Q2	Q3	Q4	Q5	Q6
az	**mDM-cpu**	11	11	1	11	11	11
	mDM-gpu	0.3	0.3	0.5	3.2	0.4	0.3
	mDM-RM	0.1	0.1	0.1	2.9	0.3	0.4
db	**mDM-cpu**	16	16	16	18	15	16
	mDM-gpu	0.3	0.3	1.0	8.2	0.6	0.7
	mDM-RM	0.1	0.1	0.3	8.2	0.5	0.7
mc	**mDM-cpu**	12	13	30	1056	57	14
	mDM-gpu	0.4	0.3	8.4	263	2.7	4.5
	mDM-RM	0.1	0.1	3.3	259	2.4	4.3

code motion significantly reduces the redundant set operations, achieving up to 6.9x speedup. It is not effective for Q1, Q3, and Q4 because the three queries do not have redundant set operations. As more threads are used for execution, the program runs faster, but the speedups are small. This is because the total workloads in these labeled queries are small.

We further generate GPU code for RM (with code motion activated). The GPU code (mRMo-gpu) runs 1.6x to 16x faster than mRMo-cpu. Interestingly, when comparing data from Table 6 and Table 7, we note that the performance of mRMo-gpu closely matches that of mGM-o for most tasks. This observation casts doubt on the practical advantage of the join-based algorithm over the basic vertex-extension on GPU.

Performance of Subgraph Counting. In some cases, the user might be only interested in the count of subgraphs without listing them. The decomposition-based algorithm in DM is most suitable for this task. For each sub-pattern, DM simply uses the basic vertex-extension backtracking algorithm to count the matching subgraphs. One potential way to improve the performance is to adopt RM to match the sub-patterns. While the idea is hard to implement in the original DM or RM system, the implementation is easy in Matcha. Table 8 shows the execution time of labeled subgraph counting with our re-implementation of DM on both CPU and GPU, and the combined algorithm (mDM-RM). We can see that Matcha easily accelerates DM by 2.2x to 53x with GPU parallelization. The combined algorithm further improves the performance on GPU by 1.8x on average.

1.7.4 Results for Clique Counting

Next, we compare GM and Matcha for 4, 5, 6-CL. We select GM as baseline because it accelerates the algorithm with an edge-pruning technique and reportedly achieves the best performance for the task among existing systems.

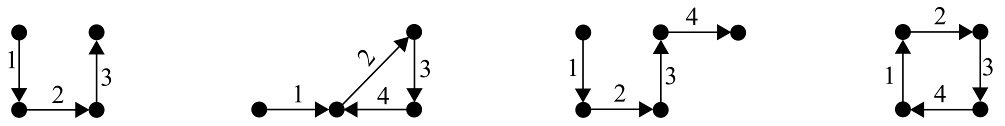
Table 9. Execution time of clique counting with the backtracking algorithm accelerated by edge pruning. 's' stands for seconds. All other numbers are in milliseconds.

		mc	**db**	**az**	**yo**	**pt**	**lj**
4-CL	**GM**	15	1.5	0.6	2.8	11	271
	mGM-o	11	0.8	0.2	1.4	8.9	271
5-CL	**GM**	592	11	0.7	5.5	13	9341
	mGM-o	551	11	0.3	4.5	10	9338
6-CL	**GM**	22.6s	235	0.7	10	17	466s
	mGM-o	20.3s	171	0.4	8.7	14	466s

Table 9 shows the execution time of the original GM system and our generated code. Our code (mGM-o) is optimized with thread-grouping and work-stealing, achieving 1.1x to 2.6x speedups with an average of 1.4x against the original GM. According to our profiling, the speedups are mainly attributed to the improved thread utilization by thread grouping. We omit the profiling data here due to the space limit.

1.7.5 Results for Temporal Subgraph Listing

We use Matcha to re-implement the edge-extension backtracking algorithm of EV and compare the performance with the original EV system.



(a) Q7 (b) Q8 (c) Q9 (d) Q10

Figure 9. Temporal Queries.

We use the four temporal queries in Fig. 9 for performance evaluation. The numbers on the edges indicate the matching order. We adopt the same setting as in the EV paper [32] and set the time constraint δ to one day (86400 seconds).

Table 10 shows the execution time of different queries with the original EV system and our generated code (mEV). Our code achieves almost the same performance as the original EV. This is because the original EV implementation already incorporates all the applicable optimizations listed in Table 1. Code motion is not helpful in this case because the edge-extension algorithm does not have any set operation, and symmetry breaking cannot be applied to temporal graphs. The results confirm the efficiency of our automatically generated code in comparison to the hand-optimized code.

Table 10. Execution time (in milliseconds) of temporal subgraph listing with edge-extension based backtracking.

		Q7	**Q8**	**Q9**	**Q10**
eu	**EV**	1.8	4.8	4.6	3.8
	mEV	1.9	5.2	4.5	4
wk	**EV**	11	21	22	20
	mEV	11	22	23	19
st	**EV**	97	190	217	185
	mEV	97	185	193	188

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[2] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29:626–688, 2015.

[3] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 739–748. IEEE, 2008.

[4] Indrajit Bhattacharya and Lise Getoor. Entity resolution in graphs. *Mining graph data*, 311, 2006.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, 2016.

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

[7] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[8] Jingji Chen and Xuehai Qian. Decomine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 47–61, 2022.

[9] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205, April 2020.

[10] Xuhao Chen et al. Efficient and scalable graph pattern mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 857–877, 2022.

[11] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159. Citeseer, 2001.

[12] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python. `https://github.com/pybind/pybind11`, 2016.

[14] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698, 2017.

[15] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[16] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. Neural subgraph matching. *arXiv preprint arXiv:2007.03092*, 2020.

[17] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–303. IEEE, 2021.

[18] Daniel Mawhirter and Bo Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523, 2019.

[19] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076*, 2019.

[20] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[21] Caleb C Noble and Diane J Cook. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2003.

[22] Steven Noel. A review of graph approaches to network security analytics. *From Database to Cyber Security*, pages 300–323, 2018.

[23] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*, pages 1431–1440, 2017.

[24] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[25] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, August 2008.

[26] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020.

[27] Johan Ugander, Lars Backstrom, and Jon Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 1307–1318, New York, NY, USA, 2013. Association for Computing Machinery.

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

[29] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. Reinforcement learning based query vertex ordering model for subgraph matching. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 245–258. IEEE, 2022.

[30] Yihua Wei and Peng Jiang. Stmatch: accelerating graph pattern matching on gpu with stack-based loop optimizations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2022.

[31] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[32] Yichao Yuan, Haojie Ye, Sanketh Vedula Wynn Kaza, and Nishil Talati. Everest: Gpu-accelerated system for mining temporal motifs. *arXiv preprint arXiv:2310.02800*, 2023.

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