Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU

Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Keshav Pingali
The University of Texas at Austin
{cxh,roshan,gill,pingali}@cs.utexas.edu

ABSTRACT

There is growing interest in graph pattern mining (GPM) problems such as motif counting. GPM systems have been developed to provide unified interfaces for programming algorithms for these problems and for running them on parallel systems. However, existing systems may take hours to mine even simple patterns in moderate-sized graphs, which significantly limits their real-world usability.

We present Pangolin, an efficient and flexible in-memory GPM framework targeting shared-memory CPUs and GPUs. Pangolin is the first GPM system that provides high-level abstractions for GPU processing. It provides a simple programming interface based on the extend-reduce-filter model, which allows users to specify application-specific knowledge for search space pruning and isomorphism test elimination. We describe novel optimizations that exploit locality, reduce memory consumption, and mitigate the overheads of dynamic memory allocation and synchronization.

Evaluation on a 28-core CPU demonstrates that Pangolin outperforms existing GPM frameworks Arabesque, RStream, and Fractal by 49×, 88×, and 80× on average, respectively. Acceleration on a V100 GPU further improves performance of Pangolin by 15× on average. Compared to state-of-the-art hand-optimized GPM applications, Pangolin provides competitive performance with less programming effort.

PVLDB Reference Format:

1. INTRODUCTION

Applications that use graph data are becoming increasingly important in many fields. Graph analytics algorithms such as PageRank and SSSP have been studied extensively and many frameworks have been proposed to provide both high performance and high productivity. Another important class of graph problems deals with graph pattern mining (GPM), which has plenty of applications in areas such as chemical engineering, bioinformatics, and social sciences. GPM discovers relevant patterns in a given graph. One example is triangle counting, which is used to mine graphs in security applications. Another example is motif counting, which counts the frequency of certain structural patterns; this is useful in evaluating network models or classifying vertex roles. Fig. 1 illustrates the 3-vertex and 4-vertex motifs.

Compared to graph analytics, GPM algorithms are more difficult to implement on parallel platforms; for example, unlike graph analytics algorithms, they usually generate enormous amounts of intermediate data. GPM systems such as Arabesque, RStream, and Fractal have been developed to provide abstractions for programmability. Instead of the vertex-centric model used in graph analytics systems, Arabesque proposed an embedding-centric programming model. In Arabesque, computation is applied on individual embeddings (i.e., subgraphs) concurrently. It provides a simple programming interface that substantially reduces the complexity of application development. However, existing systems suffer dramatic performance loss compared to hand-optimized implementations. For example, Arabesque and RStream take 98s and 39s respectively to count 3-cliques for a graph with 2.7M vertices and 28M edges, while a custom solver (KClist) counts it in 0.16s. This huge performance gap significantly limits the usability of existing GPM frameworks in real-world applications.

The first reason for this poor performance is that existing GPM systems provide limited support for application-specific customization. The state-of-the-art systems focus on generality and provide high-level abstraction to the user for ease-of-programming. Therefore, they hide as many execution details as possible from the user, which substantially limits the flexibility for algorithmic customization. The complexity of GPM algorithms is primarily due to combinatorial enumeration of embeddings and isomorphism tests to find canonical patterns. Hand-optimizing implementations exploit application-specific knowledge to aggressively prune the enumeration search space or elide isomorphism tests or both. Mining frameworks need to support such optimizations to match performance of hand-optimized applications.

The second reason for poor performance is inefficient implementation of parallel operations and data structures. Programming parallel processors requires exploring trade-offs between synchronization overhead, memory management, load balancing, and data locality. However, the state-of-the-art GPM systems target either distributed or out-of-
2. BACKGROUND AND MOTIVATION

We describe GPM concepts, applications, as well as algorithmic and architectural optimizations in state-of-the-art hand-optimized GPM solvers. Lastly, we point out performance limitations of existing GPM frameworks.

2.1 Graph Pattern Mining

In GPM problems, a pattern \( P \) is a graph defined by the user explicitly or implicitly. An explicit definition specifies the vertices and edges of the graph, whereas an implicit definition specifies the desired properties of the graph of interest. Given an input graph \( G \) and a set of patterns \( S_G \), the goal of GPM is to find the embeddings, i.e., subgraphs in \( G \) that are isomorphic to any pattern \( P \in S_G \). For explicit-pattern problems (e.g., triangle counting), the solver finds only the embeddings. For implicit-pattern problems (e.g., frequent subgraph mining), the solver needs to find the patterns as well as the embeddings. Note that graph pattern matching [50] finds embeddings only for a single explicit-pattern, whereas graph pattern mining (GPM) [3, 84] solves both explicit-pattern problems and implicit-pattern problems. In this work, we focus on connected patterns only.

In the input graph in Fig. 2, colors represent vertex labels, and numbers denote vertex IDs. The 3-vertex pattern is a blue-red-green chain, and there are four embeddings of this pattern in the input graph, shown on the right of the figure. In a specific GPM problem, the user may be interested in some pattern-specific statistical information (i.e., pattern frequency), instead of listing all the embeddings. The measure of the frequency of \( P \) in \( G \), termed support, is also defined by the user. For example, in triangle counting, the support is defined as the total count of triangles.

There are two types of GPM problems targeting two types of embeddings. In a vertex-induced embedding, a set of vertices is given and the subgraph of interest is obtained from these vertices and the set of edges in the input graph connecting these vertices. Triangle counting uses vertex-induced embeddings. In an edge-induced embedding, a set of edges is given and the subgraph is formed by including all the endpoints of these edges in the input graph. Frequent subgraph mining (FSM) is an edge-induced GPM problem.

A GPM algorithm enumerates embeddings of the given pattern(s). If duplicate embeddings exist (autormorphism), the algorithm chooses one of them as the canonical one (namely canonical test) and collects statistical information about these canonical embeddings such as the total count. The canonical test needs to be performed on each embedding, and can be complicated and expensive for complex problems such as FSM. Enumeration of embeddings in a graph grows exponentially with the embedding size (number of vertices or edges in the embedding), which is computationally expensive and consumes lots of memory. In addition, a graph isomorphism (GI) test is needed for each
embedding to determine whether it is isomorphic to a pattern. Unfortunately, the GI problem is not solvable in polynomial time \cite{37}. It leads to compute and memory intensive algorithms \cite{51} that are time-consuming to implement.

Graph analytics problems typically involve allocating and computing labels on vertices or edges of the input graph iteratively. On the other hand, GPM problems involve generating embeddings of the input graph and analyzing them. Consequently, GPM problems require much more memory and computation to solve. The memory consumption is not only proportional to the graph size, but also increases exponentially as the embedding size increases \cite{84}. Furthermore, GPM problems require compute-intensive operations, such as isomorphism test and automorphism test on each embedding. Thus, GPM algorithms are more difficult to develop, as isomorphism test and automorphism test on each embedding can be detected eagerly before extending embeddings. We term this optimization as eager pruning. Eager pruning can significantly reduce the search space.

### Pruning Enumeration Search Space

In general GPM applications, new embeddings are generated by extending existing embeddings and then they may be discarded because they are either not interesting or a duplicate (automorphism). However, in some applications like CF \cite{25}, duplicate embeddings can be detected eagerly before extending current embeddings, based on properties of the current embeddings. We term this optimization as eager pruning. Eager pruning can significantly reduce the search space. Furthermore, the input graphs are converted into directed acyclic graphs (DAGs) in state-of-the-art CF \cite{46}, CF \cite{25}, and MC \cite{71} solvers, to significantly reduce the search space.

### Eliding Isomorphism Test

Most hand-optimized GPM algorithms avoid isomorphism test by taking advantage of the pattern characteristics. For example, a parallel MC solver, PGD \cite{3}, uses an ad-hoc method for a specific k. Since it only counts 3-vertex and 4-vertex motifs, all the patterns (two 3-motifs and six 4-motifs as shown in Fig. 1) are known in advance. Therefore, some special (and thus easy-to-count) patterns (e.g., cliques) are counted first, and the frequencies of other patterns are obtained in constant time using the relationship among patterns. In this case, no isomorphism test is needed, which is typically an order-of-magnitude faster \cite{4}.

### Summary

Most of the algorithmic optimizations exploit application-specific knowledge, which can only be enabled by application developers. A generic GPM framework should be flexible enough to allow users to compose as many of these optimization techniques as possible, and provide parallelization support for ease of programming. Pangolin is the first GPM framework to do so.

### 2.3 Existing GPM Frameworks

Existing GPM systems target either distributed-memory \cite{4,5,7,6,60,34,76} or out-of-core \cite{71,81,38,82} architectures, and they make tradeoffs specific for their targeted architectures. None of them target in-memory GPM on a multicore CPU or a GPU. Consequently, they do not pay much attention to reducing the synchronization overheads among threads as well as CPU/GPU or reducing memory consumption overheads. Due to this, naive porting these GPM systems to run on a multicore CPU or GPU would lead to inefficient implementations. We first describe two of these GPM systems briefly and then discuss their major limitations.

Arabesque \cite{84} is a distributed GPM system. It proposes “think like an embedding” (TLE) programming paradigm, where computation is performed in an embedded-centric manner. It defines a filter-process computation model which consists of two functions: (1) filter, which indicates whether an embedding should be processed and (2) process, which examines an embedding and may produce some output. RStream \cite{88} is an out-of-core single-machine GPM system. Its programming model is based on relational algebra. Users specify how to generate embeddings using relational operations such as select, join, and aggregate. It stores intermediate data (i.e., embeddings) on disk while the input graph is kept in memory for reuse. It streams data (or table) from disk and uses relational operations that may produce more intermediate data, which is stored back on disk.

### Limitations in API

Most of the application-specific optimizations like pruning enumeration search space and avoiding isomorphism test are missing in existing GPM frameworks, as they focus on providing high-level abstractions but lack support for application-specific customization. The absence of such key optimizations in existing systems results in a huge performance gap when compared to hand-optimized implementations. Moreover, some frameworks like RStream support only edge-induced embeddings but for applications...
like CF, the enumeration search space is much smaller using vertex-induced exploration than edge-induced one.

Data Structures for Embeddings: Data structures used to store embeddings in existing GPM systems are not efficient. Both Arabesque and RStream store embeddings in an array of structures (AoS), where the embedding structures consist of a vertex set and an edge set. Arabesque also proposes a space efficient data structure called the Overapproximating Directed Acyclic Graph (ODAG), but it requires extra canonical test for each embedding, which has been demonstrated to be very expensive for large graphs [84].

Materialization of Data Structures: The list or array of intermediate embeddings in both Arabesque and RStream is always materialized in memory and in disk, respectively. This has significant overheads as the size of such data grows exponentially. Such materialization may not be needed if the embeddings can be filtered or processed immediately.

Dynamic Memory Allocation: As the number of (intermediate) embeddings are not known before executing the algorithm, memory needs to be allocated dynamically for them. Moreover, during parallel execution, different threads might allocate memory for embeddings they create or enumerate. Existing systems use standard (st:d) maps and sets, which internally use a global lock to dynamically allocate memory. This limits the performance and scalability.

Summary: Existing GPM systems have limitations in their API, execution model, and implementation. Pangolin addresses these issues by permitting application-specific optimizations in its API, optimizing the execution model, and providing an efficient, scalable implementation on multicore CPU and GPU. These optimizations can be applied to existing embedding-centric systems like Arabesque.

3. DESIGN OF PANGOLIN FRAMEWORK

Fig. 3 illustrates an overview of the Pangolin system. Pangolin provides a simple API (purple box) to the user for writing GPM applications. The unified execution engine (orange box) follows the embedding-centric model. Important common operations are encapsulated and provided to the user in the helper routines (blue box), which are optimized for both CPU and GPU. The embedding list data structure (green box) is also optimized for different architectures to exploit hardware features. Thus, Pangolin hides most of the architecture-oriented programming complexity and achieves high performance and high productivity simultaneously. In this section, we describe the execution model, programming interface (i.e., API), and example applications of Pangolin.

3.1 Execution Model

Algorithm 1 describes the execution engine in Pangolin which illustrates our extend-reduce-filter execution model. To begin with, a worklist of embeddings is initialized with all the single-edge embeddings (line 4). The engine then works in an iterative fashion (line 6). In each iteration, i.e., level, there are three phases: EXTEND (line 8), REDUCE (line 10), and FILTER (line 12). Pangolin exposes necessary details in each phase to enable a more flexible programming interface (Section 3.2) than existing systems; for example, Pangolin exposes the EXTEND phase which is implicit in Arabesque.

The EXTEND phase takes each embedding in the input worklist and extends it with a vertex (vertex-induced) or an edge (edge-induced). Newly generated embeddings then form the output worklist for the next level. The embedding size is increased with level until the user-defined maximum size is reached (line 14). Fig. 4 shows an example of the first iteration of vertex-based extension. The input worklist consists of all the 2-vertex (i.e., single-edge) embeddings. For each embedding in the worklist, one vertex is added to yield a 3-vertex embedding. For example, the first 2-vertex embedding \( \{0,1\} \) is extended to two new 3-vertex embeddings \( \{0,1,2\} \) and \( \{0,1,3\} \).

Algorithm 1: Execution Model for Mining

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure MineEngine(G, V, E), MAX_SIZE</td>
</tr>
<tr>
<td>2</td>
<td>EmbeddingList in_wl, out_wl $\triangleright$ double buffering</td>
</tr>
<tr>
<td>3</td>
<td>PatternMap pm $\mapsto$ stdin single-edge embeddings</td>
</tr>
<tr>
<td>4</td>
<td>intim_wl $\triangleright$ insert single-edge embeddings</td>
</tr>
<tr>
<td>5</td>
<td>level $\leftarrow$ 1</td>
</tr>
<tr>
<td>6</td>
<td>while true do $\triangleright$ clear the new worklist</td>
</tr>
<tr>
<td>7</td>
<td>out_wl $\triangleright$ $\emptyset$ $\triangleright$ clear the pattern map</td>
</tr>
<tr>
<td>8</td>
<td>EXTEND(in_wl, out_wl) $\triangleright$ clear the old worklist</td>
</tr>
<tr>
<td>9</td>
<td>pm $\mapsto$ $\emptyset$ $\triangleright$ term frequency or support, from the embedding worklist</td>
</tr>
<tr>
<td>10</td>
<td>REDUCE(out_wl, pm) $\triangleright$ filter out to collect only the frequent ones before the main loop</td>
</tr>
<tr>
<td>11</td>
<td>in_wl $\leftarrow$ $\emptyset$</td>
</tr>
<tr>
<td>12</td>
<td>FILTER(out_wl, pm, in_wl) $\triangleright$ level $\leftarrow$ level + 1</td>
</tr>
<tr>
<td>13</td>
<td>if level $=$ MAX_SIZE - 1 then</td>
</tr>
<tr>
<td>14</td>
<td>break $\triangleright$ termination condition</td>
</tr>
<tr>
<td>15</td>
<td>return in_wl, pm $\mapsto$ double buffering</td>
</tr>
</tbody>
</table>

After vertex/edge expansion, a REDUCE phase is used to extract some pattern-based statistical information, i.e., pattern frequency or support, from the embedding worklist. The REDUCE phase first classifies all the embeddings in the worklist into different categories according to their patterns, and then computes the support for each pattern category, forming pattern-support pairs. All the pairs together constitute a pattern map (pm in line 10). Fig. 5 shows an example of the reduction operation. The three embeddings (top) can be classified into two categories, i.e., triangle and wedge (bottom). Within each category, this example counts the number of embeddings as the support. As a result, we get the pattern-map as \( \{[\text{triangle}, 2], [\text{wedge}, 1]\} \). After reduction, a FILTER phase may be needed to remove those embeddings which the user are no longer interested in; e.g., FSM removes infrequent embeddings in this phase.

Note that REDUCE and FILTER phases are not necessary for all applications, and they can be disabled by the user. If they are used, they are also executed after initializing single-edge embeddings (line 4) and before entering the main loop (line 6). Thus, infrequent single-edge embeddings are filtered out to collect only the frequent ones before the main loop starts. Note that this is omitted from Algorithm 1 due to lack of space. If REDUCE is enabled but FILTER is disabled, then reduction is only required and executed for the last iteration, as the pattern map produced by reduction is not used in prior iterations (dead code).

3.2 Programming Interface

Pangolin exposes flexible and simple interfaces to the user to express application-specific optimizations. Listing 1 lists user-defined functions (APIs) and Algorithm 2 describes how these functions (marked in blue) are invoked by the Pangolin execution engine. A specific application can be created by defining these APIs. Note that all the functions are not mandatory; each of them has a default return value.

In the EXTEND phase, we provide two functions, toAdd and toExtend, for the user to prune embedding candidates aggressively. When they return false, the execution engine avoids generating an embedding and thus the search space is reduced. More specifically, toExtend checks whether...
Figure 5: Reduction operation that calculates pattern frequency using a pattern map.

**Algorithm 2** Compute Phases in Vertex-induced Mining

```c
1: procedure EXTEND(in_wel, out_wel)
2:   for each embedding emb ∈ in_wel in parallel do
3:       for each vertex v in emb do
4:         if toExtend(emb, v) = true then
5:           for each vertex u in adj(v) do
6:             if toAdd(emb, u) = true then
7:               insert emb ∪ u to out_wel

8: procedure REDUCE(queue, p_map)
9:   for each embedding emb ∈ queue in parallel do
10:      Pattern pt ← GET_PATTERN(emb)
11:     Support sp ← SUPPORT(emb)
12:   p_map[pt] ← AGGREGATE(p_map[pt], sp)

13: procedure FILTER(in_wel, p_map, out_wel)
14:   for each embedding emb ∈ in_wel in parallel do
15:      Pattern pt ← GET_PATTERN(emb)
16:     if toDiscard(pt, p_map) = false then
17:       insert emb to out_wel
```

1. bool toExtend(Embedding emb, Vertex v);
2. bool toAdd(Embedding emb, Vertex u);
3. bool toAdd(Embedding emb, Edge e);
4. Pattern getPattern(Embedding emb);
5. Support getSupport(Embedding emb);
6. Support mergeDomainSupport(Support s1, Support s2);
7. bool toDiscard(Pattern pt, PatternMap map);

**Listing 2:** Helper routines provided to the user by Pangolin.

1. // connectivity checking routines
2. 2. bool isConnected(Vertex u, Vertex v);
3. 3. // canonical test routines
4. 4. bool isAutoCanonical(Embedding emb, Vertex v);
5. 5. bool isAutoCanonical(Embedding emb, Edge e);
6. 6. Pattern get IsoCanonicalPattern(Embedding emb);
7. 7. Pattern getIsoCanonicalEigen(Embedding emb);
8. 8. // to get domain (MNI) support
9. 9. Support getDomainSupport(Embedding emb);
10. 10. Support mergeDomainSupport(Support s1, Support s2);

**uninteresting patterns. This usually depends on the support for the pattern (that is in the computed pattern map).**

**Complexity Analysis.** Consider an input graph $G$ with $n$ vertices and maximum embedding size $k$. In the **EXTEND** phase of the last level (which dominates the execution time and complexity), there are up to $O(n^{k-1})$ embeddings in the input worklist. Each embedding has up to $k-1$ vertices to extend. Each vertex has up to $d_{max}$ neighbors (candidates). In general, each candidate needs to check connectivity with $k-1$ vertices, with a complexity of $O((d_{max})^{k-1})$. An isomorphism test needs to be performed for each newly generated embedding (size of $k$) to find its pattern.

The state-of-the-art algorithm to test isomorphism has a complexity of $O(e^{\sqrt{k}\log k})$. Therefore, the overall worst-case complexity is $O(n^{k-1}k^{d_{max}}\log(d_{max})^{k-1})$.

Pangolin also provides APIs to process the embeddings or pattern maps at the end of each phase (e.g., this is used in clique-listing, which is a variant of clique-finding that requires listing all the cliques). We omit this from Algorithm 2 and Listing 1 for the sake of brevity. To implement the application-specific functions, users are required to write C++ code for CPU and CUDA versions, and an application-specific function for GPU (compiler support can provide a unified interface for both CPU and GPU in the future). Listing 2 lists the helper routines provided by Pangolin. These routines are commonly used in GPM applications, e.g., to check connectivity, test canonicity, as well as an implementation of domain support. They are available on both CPU and GPU, with efficient implementation on each architecture.

**Comparison With Other GPM APIs:** Existing GPM frameworks do not expose toExtend and getPattern to the application developer (instead, they assume these functions always return true and a canonical pattern, respectively). Note that existing embedding-centric frameworks like Arabesque can be extended to expose the same API functions in Pangolin so as to enable application-specific optimizations (Section 3.1), but this is difficult for relational model based systems like RStream, as the table join operations are inflexible to allow this fine-grained control.

### 3.3 Applications in Pangolin

TC, CF, and MC use vertex-induced embeddings, while FSM uses edge-induced embeddings. Listings 3 to 5 show CF, MC, and FSM implemented in Pangolin (we omit TC
4. SUPPORTING APPLICATION-SPECIFIC OPTIMIZATIONS IN PANGOLIN

In this section, we describe how Pangolin’s API and execution model supports application-specific optimizations that: (1) enable enumeration search space pruning and (2) enable the eliding of isomorphism tests.

4.1 Pruning Enumeration Search Space

Directed Acyclic Graph (DAG): In typical GPM applications, the input graph is undirected. In some vertex-induced GPM applications, a common optimization technique is orientation which converts the undirected input graph into a directed acyclic graph (DAG) \([24, 6]\). Instead of enumerating candidate subgraphs in an undirected graph, the direction significantly cuts down the combinatorial search space. Orientation has been adopted in triangle counting \([74]\), clique finding \([26]\), and motif counting \([71]\).

4.2 Eliding Isomorphism Test

Exploiting Memoization: Pangolin avoids redundant computation in each stage with memoization. Memoization is a tradeoff between computation and memory usage. Since GPM applications are usually memory hungry, we only do memoization when it requires small amount of memory and/or it dramatically reduce complexity. For example, in the FILTER phase of FSM, Pangolin avoids isomorphism test to get the pattern of each embedding, since it has been done in the REDUCE phase. This recomputation is avoided by maintaining a pattern ID (hash value) in each embedding after isomorphism test, and setting up a map between the pattern ID and pattern support. Compared to isomorphism test, which is extremely compute and memory intensive, storing the pattern ID and a small pattern support map is relatively lightweight. In MC, which is another application to find multiple patterns, the user can easily enable memo-
memory accesses to the embeddings are fully coalesced. This is particularly beneficial for parallel processing on GPU as GPU computing. In Pangolin, we use structure of arrays (SoA) to store embeddings in memory. The SoA layout is particularly beneficial for parallel processing on GPU as memory accesses to the embeddings are fully coalesced.

Customized Pattern Classification: In the REDUCE phase (Fig. 5), embeddings are classified into different categories based on their patterns. To get the pattern of an embedding, a generic way is to convert the embedding into a canonical graph that is isomorphic to it (done in two steps, as explained in Section 3.2). Like Arabesque and RStream, Pangolin uses the Bliss [51] library for getting the canonical graph or pattern for an embedding. This graph isomorphism approach is applicable to embeddings of any size, but it is very expensive as it requires frequent dynamic memory allocation and consumes a huge amount of memory. For small embeddings, such as 3-vertex and 4-vertex embeddings in vertex-induced applications and 2-edge and 3-edge embeddings in edge-induced applications, the canonical graph or pattern can be computed very efficiently. For example, we know that there are only 2 patterns in 3-MC (i.e., wedge and triangle in Fig. 1). The only computation needed to differentiate the two patterns is to count the number of edges (i.e., a wedge has 2 edges and a triangle has 3), as shown in Listing 6. This specialized method significantly reduces the computational complexity of pattern classification. The getPattern function in Pangolin enables the user to specify such customized pattern classification.

5. IMPLEMENTATION ON CPU AND GPU

The user implements application-specific optimizations using the Pangolin API and helper functions, and Pangolin transparently parallelizes the application. Pangolin provides an efficient and scalable parallel implementation on both shared-memory multithreaded CPU and GPU. Its GPU implementation is built using the Galois [70] library and its GPU implementation is built using the LonestarGPU [18] infrastructure. Pangolin includes several architectural optimizations. In this section, we briefly describe some of them: (1) exploiting locality and fully utilizing memory bandwidth [33, 10, 0]; (2) reducing the memory consumption; (3) mitigating the overhead of dynamic memory allocation; (4) minimizing synchronization and other overheads.

5.1 Data Structures for Embeddings

Since the number of possible $k$-embeddings in a graph increases exponentially with $k$, storage for embeddings grows rapidly and easily becomes the performance bottleneck. Most existing systems use array-of-structures ( AoS) to organize the embeddings, which leads to poor locality, especially for GPU computing. In Pangolin, we use structure of arrays (SoA) to store embeddings in memory. The SoA layout is particularly beneficial for parallel processing on GPU as memory accesses to the embeddings are fully coalesced.

Fig. 9 illustrates the embedding list data structure. On the left is the prefix-tree that illustrates the embedding extension process in Fig. 1. The numbers in the vertices are vertex IDs (VIDs). Orange VIDs are in the first level $L_1$, and blue VIDs belong to the second level $L_2$. The gray level $L_0$ is a dummy level which does not actually exist but is used to explain the key ideas. On the right, we show the corresponding storage of this prefix tree. For simplicity, we only show the vertex-induced case. Given the maximum size $k$, the embedding list contains $k−1$ levels. In each level, there are two arrays, index array $(idx)$ and vertex ID array $(vid)$. In the same position of the two arrays, an element of index and vertex ID consists of a pair $(idx, vid)$. In level $L_i$, $idx$ is the index pointing to the vertex of the same embedding in the previous level $L_{i-1}$, and $vid$ is the $i$-th vertex ID of the embedding.

Each embedding can be reconstructed by backtracking from the last level lists. For example, to get the first embedding in level $L_2$, which is a vertex set of $(0, 1, 2)$, we use an empty vertex set at the beginning. We start from the first entry $(0, 2)$ in $L_2$, which indicates the last vertex ID is ‘2’ and the previous vertex is at the position of ‘0’. We put ‘2’ into the vertex set {2}. Then we go back to the previous level $L_1$, and get the 0-th entry (0, 1). Now we put ‘1’ into the vertex set {1, 2}. Since $L_1$ is the lowest level and its index is the same as the vertex ID in level $L_0$, we put ‘0’ into the vertex set {0, 1, 2}.

For the edge-induced case, the strategy is similar but requires one more column $his$ in each level to indicate the history information. Each entry is a triplet $(vid, his, idx)$ that represents an edge instead of a vertex, where $his$ indicates at which level the source vertex of this edge is, while $vid$ is the ID of the destination vertex. In this way we can backtrack the source vertex with $his$ and reconstruct the edge connectivity inside the embedding. Note that we use three distinct arrays for $vid$, $his$ and $idx$, which is also an SoA layout. This data layout can improve temporal locality with more data reuse. For example, the first $vid$ in $L_1 (v_1)$ is connected to two vertices in $L_2 (v_2$ and $v_3$). Therefore $v_1$ will be reused. Considering high-degree vertices in power-law graphs, there are lots of reuse opportunities.

5.2 Avoiding Data Structure Materialization

Loop Fusion: Existing GPM systems first collect all the embedding candidates into a list and then call the user-defined function (like $tocado$) to select embeddings from the list. This leads to materialization of the candidate embeddings list. In contrast, Pangolin preemptively discards embedding candidates using the $tocado$ function before adding it to the embedding list (as shown in Algorithm 2), thereby avoiding the materialization of the candidate embeddings.

```c
1 if (emb.size() == 3) {
2   if (emb.getNumEdges() == 3) return P1;
3   else return P0;
4 } else return getIsoCanonicalBliss(emb);
5
6 Listing 6: Customized pattern classification for 3-MC.
```
(this is similar to loop fusion in array languages). This significantly reduces memory allocations, yielding lower memory usage and execution time.

**Blocking Schedule:** Since the memory consumption increases exponentially with the embedding size, existing systems utilize either distributed memory or disk to hold the data. However, Pangolin is a shared memory framework and could run out of memory for large graphs. In order to support processing large datasets, we introduce an edge-blocking technique in Pangolin. Since an application starts expansion with single-edge embeddings, Pangolin blocks the initial embedding list into smaller chunks, and processes all levels (main loop in Algorithm 1) for each chunk one after another. As shown in Fig. 10, there are $n$ edges in the initial embedding list ($e_0 \sim e_{n-1}$). Each chunk contains 4 edges which are assigned to the 2 threads ($t_0 \sim t_1$) to process. After all levels of the current chunk are processed, the threads move to the next chunk and continue processing until all chunks are processed. The chunk size $C_s$ is a parameter to tune; $C_s$ is typically much larger than the number of threads. Blocking will not affect parallelism because there are a large number of edges in each chunk that can be processed concurrently. Note that the FILTER phase requires strict synchronization in each level, so edge-blocking cannot be applied for applications that use it. For example, we need to gather embeddings for each pattern in FSM in order to compute the domain support. Due to this, all embeddings need to be processed before moving to the next level, so we disable blocking for FSM. Currently, edge-blocking is used specifically for bounding memory usage, but it is also potentially beneficial for data locality with an appropriate block size. We leave this for future work.

### 5.3 Dynamic Memory Allocation

**Inspection-Execution:** Compared to graph analytics applications, GPM applications need significantly more dynamic memory allocations and memory allocation could become a performance bottleneck. A major source of memory allocation is the embedding list. As the size of embedding list increases, we need to allocate memory for the embeddings in each round. When generating the embedding list, there are write conflicts as different threads write to the same shared embedding list. In order to avoid frequent resize and insert operation, we use inspection-execution technique to generate the embedding list.

The generation include 3 steps. In the first step, we only calculate the number of newly generated embeddings for each embedding in the current embedding list. We then use parallel prefix sum to calculate the start index for each current embedding, and allocate the exact amount of memory for all the new embeddings. Finally, we actually write the new embeddings to update the embedding list, according to the start indices. In this way, each thread can write to the shared embedding list simultaneously without conflicts. Fig. 11 illustrates the inspection process. At level $i$, there are $4$ embeddings $e_0, e_1, e_2, e_3$ in the embedding list, which will generate $1, 2, 3, 3$ new embeddings respectively. We get the start indices $(0, 1, 3, 4)$ using prefix sum, and then allocate memory for the level $i + 1$ embedding list. Next, each embedding writes generated embeddings from its start index in the level $i + 1$ list (concurrently).

Although inspection-execution requires iterating over the embeddings twice, making this tradeoff for GPU is reasonable for two reasons. First, it is fine for the GPU to do the recomputation as it has a lot of computation power. Second, improving the memory access pattern to better utilize memory bandwidth is more important for GPU. This is also a more scalable design choice for the CPU as the number of cores on the CPU are increasing.

**Scalable Allocators:** Pattern reduction in FSM is another case where dynamic memory allocation is frequently invoked. To compute the domain support of each pattern, we need to gather all the embeddings associated with the same pattern (see Fig. 2). This gathering requires resizing the vertex set of each domain. The C++ standard std::library employs a concurrent allocator implemented by using a global lock for each allocation, which could seriously limit performance and scalability. We leverage the Galois memory allocator to alleviate this overhead. Galois provides an in-built efficient and concurrent memory allocator that implements ideas from prior scalable allocators. The allocator uses per-thread memory pools of huge pages. Each thread manages its own memory pool. If a thread has no more space in its memory pool, it uses a global lock to add another huge page to its pool. Most allocations thus avoid locks. Pangolin uses variants of std::data structures provided by Galois that use the Galois memory allocator. For example, this is used for maintaining the pattern map. On the other hand, our GPU infrastructure currently lacks support for efficient dynamic memory allocation inside CUDA kernels. To avoid frequent resize operations inside kernels, we conservatively calculate the memory space required and pre-allocate bit vectors for kernel use. This pre-allocation requires much more memory than is actually required, and restricts our GPU implementation to smaller inputs for FSM.

### 5.4 Other Optimizations

GPM algorithms make extensive use of connectivity operations for determining how vertices are connected in the input graph. For example, in $k$-cliques, we need to check whether a new vertex is connected to all the vertices in the current embedding. Another common connectivity operation is to determine how many vertices are connected to given vertices $v_0$ and $v_1$, which is usually obtained by computing the intersection of the neighbor lists of the two vertices. A naive solution of connectivity checking is to search for one vertex $v_0$ in the other vertex $v_1$’s neighbor list sequentially. If found, the two vertices are directly connected. To reduce complexity and improve parallel efficiency, we

**Table 1:** Input graphs (symmetric, no loops, no duplicate edges) and their properties ($\bar{d}$ is the average degree).

<table>
<thead>
<tr>
<th>Graph</th>
<th>Source</th>
<th># V</th>
<th># E</th>
<th>$\bar{d}$</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Microbiot</td>
<td>1,000,000</td>
<td>2,160,312</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td>Pa</td>
<td>Patents</td>
<td>2,745,761</td>
<td>27,930,818</td>
<td>10</td>
<td>37</td>
</tr>
<tr>
<td>Yo</td>
<td>Youtube</td>
<td>7,066,392</td>
<td>114,190,484</td>
<td>16</td>
<td>29</td>
</tr>
<tr>
<td>Fdb</td>
<td>ProteinDB</td>
<td>48,738,700</td>
<td>387,730,670</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>Lj</td>
<td>LiveJournal</td>
<td>4,847,571</td>
<td>85,702,474</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Or</td>
<td>Orkut</td>
<td>3,072,441</td>
<td>234,760,166</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>Tw</td>
<td>Twitter</td>
<td>21,297,772</td>
<td>530,651,090</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Gsh</td>
<td>Gsh-2015</td>
<td>988,490,691</td>
<td>51,384,410,434</td>
<td>52</td>
<td>0</td>
</tr>
</tbody>
</table>
generalize the binary search approach proposed for TC [46] to implement connectivity check in Pangolin. This is particularly efficient on GPU, as it improves GPU memory efficiency. We provide efficient CPU and GPU implementations of these connectivity operations as helper routines, such as isConnected (Listing 2), which allow the user to easily compose pruning strategies in applications.

In summary, when no algorithmic optimization is applied, programming in Pangolin should be as easy as previous GPM systems like Arabesque. In this case, performance gains over Arabesque is achieved due to the architectural optimizations (e.g., data structures) in Pangolin. To incorporate algorithmic optimizations, the user can leverage Pangolin API functions (e.g., toExtend and toAdd) to express application-specific knowledge. While this involves slightly more programming effort, the user can get an order of magnitude performance improvement by doing so.

### 6. EVALUATION

In this section, we compare Pangolin with state-of-art GPM frameworks and hand-optimized applications. We also analyze Pangolin performance in more detail.

#### 6.1 Experimental Setup

We compare Pangolin with state-of-the-art GPM frameworks: Arabesque [84], RStream [88], G-Miner [19], Kaleido [91], Fractal [30], and AutoMine [66]. Arabesque, G-Miner, and Fractal support distributed execution, while the rest support out-of-core execution. None of them support GPU execution. Kaleido and AutoMine results are reported from their papers because they are not publicly available. We also compare Pangolin with the state-of-the-art hand-optimized GPM applications [11], [41], [20], [73], [52], [53], [54].

We test the 4 GPM applications discussed in Section 6.3, i.e., TC, CF, MC, and FSM. k-MC and k-CF terminate when subgraphs reach a size of k vertices. For k-FSM, we mine the frequent subgraphs with k – 1 edges. Table 1 lists the input graphs used in the experiments. We assume that the rest support out-of-core execution. None of them support GPU execution. Kaleido and AutoMine results are reported from their papers because they are not publicly available. We also compare Pangolin with the state-of-the-art hand-optimized GPM applications [11], [41], [20], [73], [52], [53], [54].

We test the 4 GPM applications discussed in Section 6.3, i.e., TC, CF, MC, and FSM. k-MC and k-CF terminate when subgraphs reach a size of k vertices. For k-FSM, we mine the frequent subgraphs with k – 1 edges. Table 1 lists the input graphs used in the experiments. We assume that the rest support out-of-core execution. None of them support GPU execution. Kaleido and AutoMine results are reported from their papers because they are not publicly available. We also compare Pangolin with the state-of-the-art hand-optimized GPM applications [11], [41], [20], [73], [52], [53], [54].

We test the 4 GPM applications discussed in Section 6.3, i.e., TC, CF, MC, and FSM. k-MC and k-CF terminate when subgraphs reach a size of k vertices. For k-FSM, we mine the frequent subgraphs with k – 1 edges. Table 1 lists the input graphs used in the experiments. We assume that the rest support out-of-core execution. None of them support GPU execution. Kaleido and AutoMine results are reported from their papers because they are not publicly available. We also compare Pangolin with the state-of-the-art hand-optimized GPM applications [11], [41], [20], [73], [52], [53], [54].

We test the 4 GPM applications discussed in Section 6.3, i.e., TC, CF, MC, and FSM. k-MC and k-CF terminate when subgraphs reach a size of k vertices. For k-FSM, we mine the frequent subgraphs with k – 1 edges. Table 1 lists the input graphs used in the experiments. We assume that the rest support out-of-core execution. None of them support GPU execution. Kaleido and AutoMine results are reported from their papers because they are not publicly available. We also compare Pangolin with the state-of-the-art hand-optimized GPM applications [11], [41], [20], [73], [52], [53], [54].

### 6.2 GPM Frameworks

Table 2 reports the execution time of Arabesque, RStream, Kaleido, Fractal, and Pangolin. The execution time of G-Miner and AutoMine is reported in Table 3b and Table 4 respectively (because it does not have other applications or datasets respectively). Note that Kaleido and AutoMine results on 28-core and 20-core CPU, respectively, are reported from their papers. We evaluate the rest on our 28-core CPU, except that we evaluate Pangolin for gsh on 48-core CPU. Fractal and AutoMine use DFS exploration [59, 50], whereas the rest use BFS. Pangolin is an order-of-magnitude faster than Arabesque, RStream, Fractal, and G-Miner. Pangolin outperforms Kaleido in all cases except 4-MC on patent. Pangolin on CPU is comparable or slower than AutoMine but outperforms it by exploiting the GPU.

For small inputs (e.g., TC and 3-CF with Mi), Arabesque suffers non-trivial overhead due to the startup cost of GiRaph. For large graphs, however, due to lack of algorithmic (e.g., eager pruning and customized pattern classification) and data structure optimizations, it is also slower than Pangolin. On average, Pangolin is 49× faster than Arabesque.

For RStream, the number of partitions P is a key performance knob. For each configuration, we choose P to be the

<table>
<thead>
<tr>
<th>Option</th>
<th>AR</th>
<th>RS</th>
<th>KA</th>
<th>PA</th>
<th>FR</th>
<th>PA</th>
<th>AR</th>
<th>RS</th>
<th>KA</th>
<th>FR</th>
<th>PA</th>
<th>AR</th>
<th>RS</th>
<th>KA</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>3</td>
<td>32.2</td>
<td>7.3</td>
<td>0.5</td>
<td>24.7</td>
<td>0.02</td>
<td>190.8</td>
<td>7.8</td>
<td>0.5</td>
<td>0.08</td>
<td>601.3</td>
<td>39.8</td>
<td>2.2</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>CF</td>
<td>4</td>
<td>41.7</td>
<td>637.8</td>
<td>3.9</td>
<td>30.6</td>
<td>1.6</td>
<td>108.1</td>
<td>62.1</td>
<td>1.1</td>
<td>410.1</td>
<td>0.4</td>
<td>1008.9</td>
<td>- 7.8</td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>MC</td>
<td>5</td>
<td>311.9</td>
<td>- 183.6</td>
<td>488.9</td>
<td>60.5</td>
<td>108.8</td>
<td>76.9</td>
<td>1.5</td>
<td>463.5</td>
<td>0.5</td>
<td>1123.6</td>
<td>- 19.0</td>
<td>19.0</td>
<td>7.3</td>
<td></td>
</tr>
<tr>
<td>3-FSM</td>
<td>3</td>
<td>36.1</td>
<td>1737.5</td>
<td>1.4</td>
<td>41.2</td>
<td>0.2</td>
<td>101.6</td>
<td>3886.9</td>
<td>1.7</td>
<td>236.3</td>
<td>0.9</td>
<td>358.4</td>
<td>89387.0</td>
<td>35.5</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>353.0</td>
<td>- 198.2</td>
<td>243.2</td>
<td>175.6</td>
<td>779.8</td>
<td>- 152.3</td>
<td>561.1</td>
<td>209.1</td>
<td>5132.8</td>
<td>- 4989.0</td>
<td>4405.3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unless specified otherwise, CPU experiments were conducted on a single machine with Intel Xeon Gold 5120 CPU 2.2GHz, 4 sockets (14 cores each), 190GB memory, and 3TB SSD. AutoMine was evaluated using 40 threads (with hyper-threading) on Intel Xeon E5-2630 v4 CPU 2.2GHz, 2 sockets (10 cores each), 64GB of memory, and 2TB of SSD. Kaleido was tested using 56 threads (with hyper-threading) on Intel Xeon Gold 5117 CPU 2.0GHz, 2 sockets (14 cores each), 128GB memory, and 480GB SSD. To make our comparison fair, we restrict our experiments to use only 2 sockets of our machine, but we only use 28 threads without hyper-threading.

For the largest graph, Gsh, we used a 2 socket machine with Intel’s second generation Xeon scalable processor with 2.2 Ghz and 48 cores, equipped with 6TB of Intel Optane PMM (byte-addressable memory technology). Our GPU platforms are NVIDIA GTX 1080Ti (11GB memory) and Tesla V100 (32GB memory) GPUs with CUDA 9.0. Unless specified otherwise, GPU results reported are on V100.

RStream writes its intermediate data to the SSD, whereas other frameworks run all applications in memory. We exclude preprocessing time and only report the computation time (on the CPU or GPU) as an average of 3 runs. We also exclude the time to transfer data from CPU to GPU as it is trivial compared to the GPU compute time.
Table 3: Execution time (sec) of Pangolin (PA) and hand-optimized solvers (σ; minimum support). PA-GPU and DistTC-GPU are on V100 GPU; PGD-GPU is on Titan Black GPU; rest are on 28-core CPU. 1PGD-GPU results are reported from their paper.

(a) TC: GM: G-Miner. (b) 4-CF. (c) 3-MC.

<table>
<thead>
<tr>
<th>Application</th>
<th>PA-CPU</th>
<th>PA-GPU</th>
<th>DistTC-GPU</th>
<th>PA-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>9.9</td>
<td>4.2</td>
<td>3.9</td>
<td>4.3</td>
</tr>
<tr>
<td>3-MC</td>
<td>1.17</td>
<td>0.31</td>
<td>0.25</td>
<td>0.42</td>
</tr>
<tr>
<td>4-MC</td>
<td>2.74</td>
<td>0.43</td>
<td>0.43</td>
<td>0.73</td>
</tr>
<tr>
<td>5-CF</td>
<td>3.15</td>
<td>0.60</td>
<td>0.62</td>
<td>1.43</td>
</tr>
</tbody>
</table>

Table 4: Execution time (sec) of Pangolin (PA) and AutoMine (AM). Pangolin for Gsh is evaluated on Intel Optane-PMM machine. 1AutoMine results are reported from its paper.

<table>
<thead>
<tr>
<th>Application</th>
<th>PA-CPU</th>
<th>PA-GPU</th>
<th>DistTC-GPU</th>
<th>PA-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>4966</td>
<td>139.3</td>
<td>129.0</td>
<td>4.3</td>
</tr>
<tr>
<td>3-MC</td>
<td>0.20</td>
<td>59.3</td>
<td>65.9</td>
<td>3.9</td>
</tr>
<tr>
<td>4-MC</td>
<td>0.43</td>
<td>35.3</td>
<td>34.3</td>
<td>3.9</td>
</tr>
<tr>
<td>5-CF</td>
<td>9.7</td>
<td>69.5</td>
<td>63.7</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table 5: Lines of code in Pangolin (PA) and hand-optimized (HO) applications (implementation name in parenthesis).

<table>
<thead>
<tr>
<th>Application</th>
<th>PA-CPU</th>
<th>PA-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>26</td>
<td>36</td>
</tr>
<tr>
<td>3-MC</td>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td>4-MC</td>
<td>82</td>
<td>92</td>
</tr>
<tr>
<td>5-CF</td>
<td>252</td>
<td>234</td>
</tr>
</tbody>
</table>

best performing one among 10, 20, 50, and 100. RStream only supports edge-induced exploration and does not support pattern-specific optimization. This results in extremely large search spaces for CF and MC because there are many more edges than vertices. In addition, RStream does not scale well because of the intensive use of mutex locks for updating shared data. Lastly, Pangolin avoids inefficient data structures and expensive redundant computation (isomorphism test) used by RStream. Pangolin is 88× faster than RStream on average (Kaleido [91] also observes that RStream is slower than Arabesque).

On average, Pangolin is 2.6× faster than Kaleido (7.4×, 3.3×, 2.4×, and 1.6× for TC, CF, MC, and FSM respectively). This is mainly due to DAG construction and customized pattern classification in Pangolin.

Pangolin is on average 80× faster than Fractal. Fractal is built on Spark and suffers from overheads due to it. More importantly, some optimizations in hand-optimized DFS-based applications like PGD [4] and KClist [26] are not supported in Fractal, which limits its performance.

AutoMine uses a key optimization [4, 26] to remove redundant computation that can only be enabled in DFS-based exploration. Due to this, when pattern size k is large like in 5-CF and 4-MC, AutoMine is faster than Pangolin. However, since Pangolin uses BFS-based exploration which easily enables GPU acceleration, Pangolin on GPU is on average 5.8× faster than AutoMine. It is not clear how to enable DFS mode for GPU efficiently, especially when k is large. Note that for all the applications, AutoMine can only do counting but not listing, because it has no automorphism test during extension (instead it uses post-processing to address the multiplicity issue). FSM in AutoMine uses frequency (which is not anti-monotonic) instead of domain support, and thus it is not comparable to FSM in Pangolin.

6.3 Hand-Optimized GPM Applications

We compare hand-optimized implementations with Pangolin on CPU and GPU. We report results for the largest datasets supported on our platform for each application. Note that all hand-optimized applications involve substantially more programming effort than Pangolin ones. As shown in Table 3, hand-optimized TC has 4× more lines of code (LoC) than Pangolin TC and the other hand-optimized applications have one or two orders of magnitude more LoC than Pangolin ones. The Pangolin code for MC is shown in Listings 1 and 9. The lines in the other Pangolin applications are as simple as that in MC. Hand-optimized solvers must handle parallelism, synchronization, memory allocation, etc, while Pangolin transparently handles all of that, making it easier for the user to write applications.

In Table 3a, we compare with GAP [11] and DistTC [44], the state-of-the-art TC implementations on CPU and GPU, respectively. It is clear from Table 3a and Table 3b that TC implementations in existing GPM frameworks are orders of magnitude slower than the hand-optimized implementation in GAP. In contrast, Pangolin performs similar to GAP on the same CPU. Pangolin is also faster than DistTC on the same GPU due to its embedding list data structure, which has better load balance and memory access behavior.

Table 3c compares our 4-clique with KClist [26], the state-of-the-art CF implementation. Pangolin is 10 to 20× slower than KClist on the CPU, although GPU acceleration of Pangolin significantly reduces the performance gap. This is because KClist constructs a shrinking local graph for each edge, which significantly reduces the search space. This optimization can only be enabled in the DFS exploration. In Table 3d, we observe the same trend for 3-MC compared with PGD, the state-of-the-art MC solver for multicore CPU [4] and GPU [73]. Note that PGD can only do counting, but not listing, as it only counts some of the patterns and the other patterns’ counts are calculated directly using some formulas. In contrast, MC in Pangolin can do both counting and listing. Another limitation of PGD is that it can only handle 3-MC and 4-MC, while Pangolin handles arbitrary k. As PGD for GPU (PGD-GPU) [73] is not released, we estimate PGD-GPU performance using their reported speedup [73] on Titan Black GPU. Pangolin-GPU is 20% to 130% slower.

Table 3e and Table 3f compares our 3-FSM and 4-FSM, respectively, with DistGraph [22, 31]. DistGraph supports both shared-memory and distributed platforms. DistGraph supports a runtime parameter σ, which specifies the minimum support, but we had to modify it to add the maximum size k. On GPU, Pangolin outperforms DistGraph for 3-FSM in all cases, except for Pa with support 5K. For graphs that fit in the GPU memory (Mi, Pa), Pangolin on GPU is 6.9× to 290× faster than DistGraph. In comparison, the GPU implementation of DistGraph is only 4× to 9× faster than its CPU implementation [52] (we are not able to run their GPU code and we cannot compare.
with their reported results as they do not evaluate the same datasets). For 4-FSM, Pangolin is 22% to 240% slower than DistGraph. The slowdown is mainly due to the algorithmic differences: DistGraph adopts DFS exploration and a recursive approach which reduces computation and memory consumption, while Pangolin does BFS exploration.

6.4 Scalability and GPU Performance

Although Pangolin is an in-memory processing system, Pangolin can scale to very large datasets by using large memory systems. To demonstrate this, we evaluate Pangolin on the Intel Optane PMM system and mine a very large real-world web crawl, Gash. As shown in Table 3, TC and 3-CF only take 2 and 11 minutes, respectively. 4-CF is much more compute and memory intensive, so it takes $\sim 6.5$ hours.

Fig. 13 illustrates how the performance of Pangolin applications scales as the number of threads increases for different applications on Yo. Pangolin achieves good scalability by utilizing efficient, concurrent, scalable data structures and allocators. For TC, we observe near linear speedup over single-thread execution. In contrast, FSM’s scalability suffers due to the overheads of computing domain support.

To test weak scalability, we use the RMAT graph generator to generate graphs with vertices $|V| = 2^n$ (n = 20, 21, 22, 23, 24, 25) and average degree $d = 20$. Fig. 13 reports the execution time normalized to that of rmat20 (log-log scale). The execution time grows exponentially as the graph size increases because the enumeration search space grows exponentially.

Fig. 13 illustrates speedup of Pangolin applications on GPU over 28-core CPU. Note that due to the limited memory size, GPUs fail to run some applications and inputs. On average, 1080Ti and V100 GPUs achieve a speedup of 6× and 15× respectively over the CPU. Specifically, we observe substantial speedup on CF and MC. For example, the V100 GPU achieves 50× speedup on 4-MC for Yo, demonstrating the suitability of GPUs for these applications.

6.5 Memory Consumption

The peak memory consumption for Arabesque, RStream, and Pangolin on the same 28-core CPU platform is illustrated in Fig. 15. We observe that Arabesque always requires the most memory because it is implemented in Java using Giraph which allocates a huge amount of memory. In contrast, Pangolin avoids this overhead and reduces memory usage. Since Pangolin does in-memory computation, it is expected to consume much more memory than RStream which stores its embeddings in disk. However, we find that the difference in memory usage is trivial because aggressive search space pruning and customized pattern classification significantly reduce memory usage. Since this small memory cost brings substantial performance improvement, we believe Pangolin makes a reasonable trade-off.

6.6 Impact of Optimizations

We evaluate the performance improvement due to the optimizations described in Section 3 and Section 4. Due to lack of space, we present these comparisons only for the CPU implementations, but the results on the GPU are similar. Fig. 16a shows the impact of orientation (DAG) and user-defined eager pruning (Prune) on 4-CF. Both techniques significantly improve performance for TC (not shown) and CF. Fig. 16b demonstrates the advantage of using Galois memory allocators instead of std allocators. This is particularly important for FSM as it requires intensive memory allocation for counting support. Fig. 16c illustrates that customized pattern classification used in MC and FSM yields huge performance gains by eliding expensive generic isomorphism tests. Fig. 16d shows that materialization of temporary embeddings causes 11% to 37% slowdown for MC. This overhead exists in every application of Arabesque (and RStream), and is avoided in Pangolin. In Fig. 17a, we evaluate the performance of our proposed embedding list data structure with SoA layout and inspection-execution. Compared to the straight-forward embedding queue (mimic the AoS implementation used in Arabesque and RStream), the k-MC performance is 2.1× to 4.7× faster. Another optimization is employing binary search for connectivity check. Fig. 17b shows that binary search can achieve up to 6.6× speedup compared to linear search. Finally, Fig. 17c illustrates the last level cache (LLC) miss counts in the vertex extension phase of k-CF. We compare two data structure schemes for the embeddings, AoS and SoA. We observe a sharp reduction of LLC miss count by switching from AoS to SoA. This further confirms that SoA has better locality than AoS, due to the data reuse among embeddings.

7. RELATED WORK

GPM Applications: Hand-optimized GPM applications target various platforms. For triangle counting, Shun et al. [79] present a parallel, cache-oblivious TC solver on multi-core CPUs that achieves good cache performance without fine-tuning cache parameters. TriCore [46] is a multi-GPU TC solver that uses binary search to increase coalesced memory accesses, and it employs dynamic load balancing. There are several distributed TC solvers [81, 38, 74] too.

Chiba and Nishizeki (C&N) [24] proposed an efficient k-clique listing algorithm which computes the subgraph induced by neighbors of each vertex, and then recurses on the subgraph. Danisch et al. [26] refine the C&N algorithm for parallelism and construct DAG using a core value based ordering to further reduce the search space. PGD [4] counts 3 and 4-motifs by leveraging a number of proven combinatorial arguments for different patterns. Some patterns (e.g., cliques) are counted first, and the frequencies of other patterns are obtained in constant time using these combinatorial arguments. Escape [71] extends this approach to 5-vertex subgraphs and leverages DAG to reduce search space.

gSpan [90] is an efficient sequential FSM solver which does depth-first search (DFS) based on a lexicographic order. GraMi [22] proposes an approach that finds only the minimal set of instances to satisfy the support threshold and avoids enumerating all instances. DistGraph [82] parallelizes gSpan for both shared-memory and distributed CPUs. Each
worker thread does the DFS walk concurrently. It introduces a customized dynamic load balancing strategy which splits tasks on the fly and recomputes the embedding list from scratch after the task is sent to a new worker. Scalemine [1] solves FSM with a two-phase approach, which approximates frequent subgraphs in phase-1, and uses collected information to compute the exact solution in phase-2. There are other GPM applications, e.g., maximal cliques [21], maximum clique [63, 2], and subgraph listing [77, 14, 54, 49, 55, 64]. All the above hand-optimized solvers employ various optimizations to reduce computation and improve hardware efficiency. However, they achieve high performance at the cost of tremendous programming efforts, while Pangolin provides a unified model for ease of programming.

GPM Frameworks: For ease-of-programming, GPM systems such as Arabesque [84], RStream [88], G-Miner [19], and Kaleido [81] have been proposed. They provide a unified programming interface to the user which simplifies application development. However, their interface is not flexible enough to enable application specific optimizations. Instead of the BFS exploration used in these frameworks, Fractal [90] employs a DFS strategy to enumerate subgraphs, which reduces memory footprint. AutoMine [69] is a compiler based system using BFS exploration. In contrast, Pangolin uses the BFS approach that is inherently more load-balanced, and is better suited for GPU acceleration.

Approximate GPM: There are approximate solvers for TC [86, 72, 85], CF [69, 48], MC [80, 16], and FSM [7].

8. CONCLUSION
We present Pangolin, a high-performance, flexible GPM system on shared-memory CPUs and GPUs. Pangolin provides a simple API that enables the user to specify eager enumeration search space pruning and customized pattern classifications. To exploit locality, Pangolin uses an efficient structure of arrays (SoA) for storing embeddings. It avoids materialization of temporary embeddings and blocks the schedule of embedding exploration to reduce the memory usage. It also uses inspection-execution and scalable memory allocators to mitigate the overheads of dynamic memory allocation. These application-specific and architectural optimizations enable Pangolin to outperform prior GPM frameworks, Arabesque, RStream, and Fractal, by 49×, 88×, and 80×, on average, respectively, on the same 28-core CPU. Moreover, Pangolin on V100 GPU is 15× faster than that on the CPU on average. Thus, Pangolin provides performance competitive with hand-optimized implementations but with much better programming experience.

Acknowledgments
The research was supported by NSF grants 1406355, 1618425, 1705902, and 1725322; NSFC grant 61802416; and DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. We thank Intel for providing the Intel Optane DC PMM machine.
9. REFERENCES


