FAST: FPGA-based Subgraph Matching on Massive Graphs

Xin Jin[†], Zhengyi Yang[§], Xuemin Lin[§], Shiyu Yang[†], Lu Qin[‡], You Peng[§]

[†]*East China Normal University,* [§]*University Of New South Wales,* [‡]*University of Technology Sydney*

xinjin@stu.ecnu.edu.cn, {zyang,lxue}@cse.unsw.edu.au,
syyang@sei.stu.ecnu.edu.cn, lu.qin@uts.edu.au, you.peng@unsw.edu.au

We would like to thank the reviewers for their insightful and invaluable comments. We have revised the paper carefully according to the comments. A summary of major revisions is given below, followed by the point-to-point response to each reviewer.

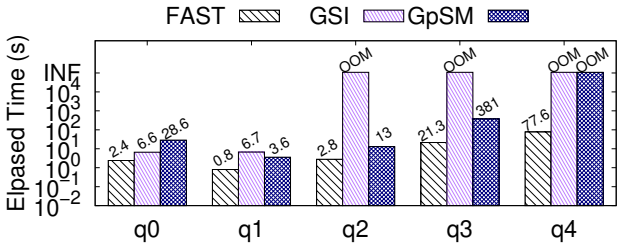
1) Add additional comparison with GPU-based solutions.
2) Provide the statistics about the number and size of CST to show the scalability of the partition mechanism.
3) Conduct more experiments as suggested by reviewers for better comparison.
4) Strengthen the application of subgraph matching on FPGAs.
5) Revise the figures, proof-read and improve presentation.

Please kindly find our point-to-point responses below.

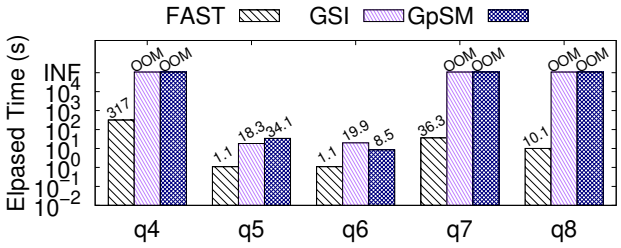
RESPONSE TO REVIEWER #1

Comment 1.1 **(R1)** Compare with GPU based solutions

Response. Thanks. We have added experiments in the revised paper to compare FAST with two state-of-the-art GPU-based solutions GSI [39] and GpSM [35]. We do not compare PBE [16] because it is unable to handle labeled graphs. The results are illustrated in *Comparing with Existing Algorithms* (Section VII-C, Fig. 12). For demonstration, we extract the results of FAST compared with GSI and GpSM and present them here in Fig. 1 (q_0-q_4 in DG03 and q_4-q_8 in DG10 due to space limit). As shown, FAST outperforms GSI and GpSM for all the queries. Note that both GSI and GpSM fail to complete all the queries due to out of memory (denoted as OOM).



(a) DG03



(b) DG10

Fig. 1. Elapsed time of FAST, GSI and GpSM

Comment 1.2 **(R2)** Rewrite the text of the System Overview scetion including symbol tables in the Software Implementation subsection and making sure all terms are concretely defined when used.

Response: Thanks. We have carefully rewritten the text of the *Software Implementation* (Section V). A symbol table (Table 1) is added at the beginning of Section V. Specifically, we have made the following changes with respect to W1:

- We modify the data graph G in the Fig. 1(b) and CST in the Fig. 3(b) in the revised paper so that the Example 2 can be easier to understand.
- We define the unclear symbols (e.g. $\mathcal{O}$) and terms (e.g. 'valid node') before used.
- CST is not a DAG nor a tree. It is an undirected graph as given in Definition 2. We abuse the term *tree* during naming to emphasize that CST is constructed based on the spanning tree of the query graph following conventions. A footnote has been added in Definition 2 to clarify.
- Each vertex in CST has a candidate set in which each candidate in it refers to a data vertex in the data graph. We have modified the definition of CST (Definition 2) to make it clear. In addition, we use the term 'candidate' to distinguish a candidate vertex from a CST vertex after Definition 2 to remove ambiguity.
- $N^u_{u'}(v)$ is the adjacency list of $v \in C(u)$ with respect to (u, u') in CST, i.e., $N^u_{u'}(v) = \{v' \in C(u') \mid (v, v') \in E(\text{CST})\}$. The definition is added in the third paragraph of Section V-A.
- We have added CSTProcess (Algorithm 3) to clarify the scheduling process.

Comment 1.3 **(R3)** Record some statistics about the number of partitions that was used in each experiment

as well as the total size of the CST.

Response. Thanks. We have added the statistics about the number of partitions and the total size of CST in the revised paper. The results are illustrated in *The Necessity of CST Partition* (Section VII-A, Fig. 8).

Our partition method demonstrates good scalability in the experiments. Let S_{CST} and S_G be the size of all CST partitions and data graph, respectively. In general, $\frac{S_{\mathsf{CST}}}{S_G} < 60\%$ for all experiments and $\frac{S_{\mathsf{CST}}}{S_G}$ keeps stable for the most of queries while the data graph grows. The rapid growth of $\frac{S_{\mathsf{CST}}}{S_G}$ in q_7 from DG03 to DG10 is due to the rapid increase in the number of embeddings.

Moreover, we evaluate the partition time with respect to the number of embeddings as the data graph grows. The results are shown in Fig. 2. The partition time is almost linear to the number of embeddings, with the average partition time increases only slightly (1.09×10^{-9}, 1.15×10^{-9}, 2.11×10^{-9}, and 2.15×10^{-9} seconds per embedding for DG01, DG03, DG10 and DG60, respectively), while the sizes of data graphs grows rapidly (the numbers of edges are 17.24M, 52.65M, 176.48M and 1.25B for DG01, DG03, DG10 and DG60, respectively). This proves the scalability of our partition mechanism when the data graph grows.

Subgraph matching on complex query in large data graphs is challenging for all algorithms. The queries we use are selected from the *Interactive Workload* in the LDBC-SNC benchmark which already includes complex queries in practical (e.g. the number of results of q_7 in DG60 exceeds 4 billion). The results show that our partition mechanism can handle complex real-world workloads.

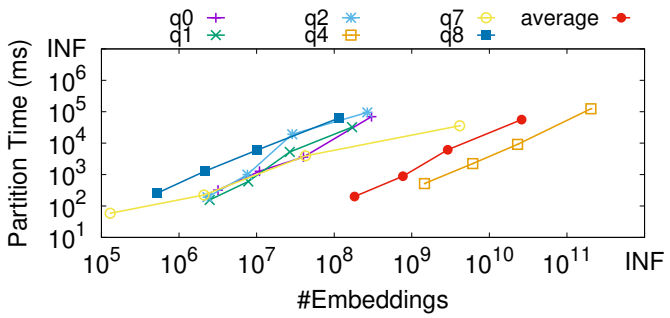


Fig. 2. The partition time per embedding

Comment 1.4 **(D4)** The logarithmic scale used in the experiments is not ideal for acceleration ratios < 1.

Response. Thanks. In the revised version, we label the actual acceleration ratio on the right side of the figures.

Response to Reviewer #2

Comment 2.1 (D3) The claims of novelty of the CST data structure should be toned down. Transforming the query graph into a spanning tree and matching the edges in the tree first is a well-known technique used by many algorithms. CST might not be exactly the same as data structures used by previous work, but it is a minor variations. A tree partitioning technique seemingly similar to the one of this paper was already introduced by the QFrag algorithm.

Response. Thanks. We have revised the description in the first paragraph of CST *Structure* (Section V-A) to lower our tone. CST is similar to the auxiliary data structure in previous works, but with vital difference. The design choice is discussed at the end of Section V-A in the Remark. We refer to the main contribution as our CPU-FPGA co-designed framework with the help of CST structure and its partition mechanism, instead of solely the design of CST.

In addition, the partitioning technique of CST is different with the one of QFrag [30]: (1) QFrag partitions its tree-like structure only on the second query vertex in the matching order. In FAST, we partition CST from the first query vertex to the last one until CST meets the requirements. (2) QFrag partitions its tree-like structure for load-balancing, so the partition factor is fixed (equals to the number of workers). The partition factor of CST is dynamically determined each round. (3) QFrag does not partition the relationships between the non-tree neighbors like us.

Comment 2.2 (D5) Figure 11 has a lot of information and its y axis is in log-scale. It is difficult to understand the exact speedup for the different experiments, and the text only reports the best-case speedups. Please add numbers on top of each bar to make the figure more readable.

Response. Thanks. We have fixed Fig. 11 (Fig. 12 now) in the revised paper. Each query shows similar trend in different data graphs, so we choose to present the results of only five queries for each data graph to make the figure easy to read. The complete results is included in our technical report [21].

Response to Reviewer #3

Comment 3.1 (D1.1) While it is understood CST partitioning is necessary, the choice of k (Lines 2-3 of Algo. 2) requires some explanation or experiments. Is the acceleration sensitivity to k?

Response. Thanks. The acceleration is not very sensitive to k when k is in asmall range (e.g. $k \leq 10$). The choice of k do make impact on the partition time, but when k is small, the partition time of CPU can overlap well with the time of computing matchings on FPGA (the most time-consuming process). When k is large, the partition time can potentially increase and hence slow down the acceleration. However, our greedy strategy can select a good k to reduce the time for

partitioning and the final number of CST partitions, so it can make less impact on the total computation.

We have added the k-Determination experiment as illustrated in Fig. 3. Besides our greedy strategy, we test FAST with fixed $k \in \{2, 4, 6, 8, 10\}$. The average number of CST and the average partition time are reported. It can be seen that our greedy approach does achieve the least number of CST and least time cost to partition CST.

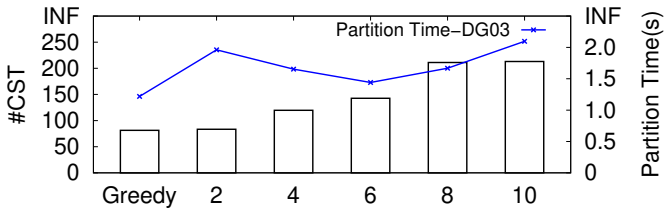


Fig. 3. The average number of CST and average partition time varying k

Comment 3.2 (D1.2) The matching $\mathcal{O}$ appears to play an important role in efficiency. It should be tested with some alternatives.

Response: Thanks. We have tested FAST with the following matching orders: (1) CFL's order (order the root-to-leaf path of a spanning tree). (2) DAF's order (a topological order of DAG). (3) CECI's order (a breadth-first search order). (4) all random connected orders. The results are added in the revised paper (Section VII-C, Fig. 13).

For each query, we extract the minimum, average and maximum time in all random connected orders as the results of BEST, AVG and WORST matching orders, respectively. It can be seen from the figure that the average elapsed time of FAST with CFL's, DAF's and CECI's matching orders is very close to each other. The FAST with the WORST matching order (about 5x slower than the BEST order in average) can still outperform CPU-based solutions CFL, CECI and DAF (by 9.6x, 11.1x and 36.3x, respectively) which further proves the effectiveness our CPU-FPGA co-designed framework.

It is practically infeasible to always compute the best plan [33], so FAST is designed to accept any matching order. In the paper, we adopt the path-ordering matching order similar to CFL-Match which present good performance in most cases.

Comment 3.3 (D1.3) Similarly, the threshold δ should be backed up by experiments.

Response. Thanks. We add the δ experiments in *Effectiveness of Software Scheduler* (Section VII-B). The results are illustrated in Fig. 11 in the revised paper. According to the figure, *Software Scheduler* optimization achieves best improvements when $\delta = 0.1$.

Comment 3.4 (D2) Are the experiments on the existing work [13, 17, 12] run on a setup that only one thread is allowed? It makes more sense if the code is compiled with its best optimization options on the host machine.

Response. Thanks. CFL [13] is designed and implemented only for single thread. DAF [17] and CECI [12] provide the parallel version of their algorithms. We run DAF and CECI on using 8 threads (on a 8-core machine) denoted as DAF-8 and CECI-8, respectively, and compare the results with FAST. However, DAF-8 encounters out of memory error when processing DG03 and DG10. So we only report the elapsed time of CECI-8 in Fig. 12 in the revised paper. We extract the elapsed time of FAST, CECI and CECI-8, as shown in the Fig. 4 here. FAST outperforms CECI-8 in all queries even when 8 threads are used, the average acceleration rate of FAST compared with CECI-8 is 5.79x, 8.51x and 9.31x for DG01, DG03 and DG10, respectively.

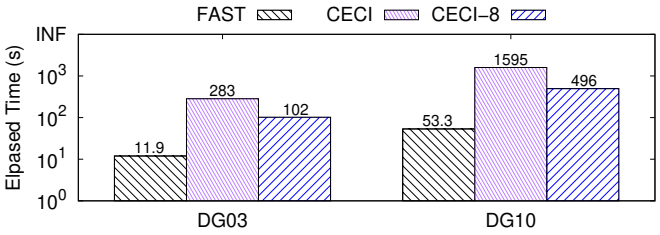


Fig. 4. The average elapsed time of CECI-8, CECI, FAST-8 and FAST

Comment 3.5 (R1) Discuss, analyze or show the parameters used in the experiments as detailed as possible (D1-D2).

Response. Thanks. We have given the corresponding response in Comment 3.1-3.4.

Comment 3.6 (R2) Strengthen the applications of the paper (D3).

Response. Thanks. Subgraph matching has a wide range of applications as mentioned in the first paragraph of the introduction. FPGA-based subgraph matching can speed up and benefit all applications of them. Moreover, subgraph matching is essentially the core operation of subgraph queries in graph databases (e.g. Neo4j) and RDF engines(e.g. gStore). FPGA-based subgraph matching can be integrated into existing graph databases and RDF engines to accelerate many real-world applications. Such attempts has already been made using GPUs but not FPGAs. We have added a short discussion in our revised paper (Section I).

Comment 3.7 (R3) Proofread the paper.

Response. Thanks. We have proofread the paper to fix the typos and mistakes we found in the paper.

FAST: FPGA-based Subgraph Matching on Massive Graphs

Xin Jin[†], Zhengyi Yang[§], Xuemin Lin[§], Shiyu Yang[†], Lu Qin[‡], You Peng[§]

[†]*East China Normal University,* [§]*University Of New South Wales,* [‡]*University of Technology Sydney*

xinjin@stu.ecnu.edu.cn, {zyang,lxue}@cse.unsw.edu.au,
syyang@sei.stu.ecnu.edu.cn, lu.qin@uts.edu.au, you.peng@unsw.edu.au

***Abstract*—Subgraph matching is a basic operation widely used in many applications. However, due to its NP-hardness and the explosive growth of graph data, it is challenging to compute subgraph matching, especially in large graphs. In this paper, we aim at scaling up subgraph matching on a single machine using FPGAs. Specifically, we propose a CPU-FPGA co-designed framework. On the CPU side, we first develop a novel auxiliary data structure called candidate search tree (CST) which serves as a complete search space of subgraph matching. CST can be partitioned and fully loaded into FPGAs' on-chip memory. Then, a workload estimation technique is proposed to balance the load between the CPU and FPGA. On the FPGA side, we design and implement the first FPGA-based subgraph matching algorithm, called FAST. To take full advantage of the pipeline mechanism on FPGAs, task parallelism optimization and task generator separation strategy are proposed for FAST, achieving massive parallelism. Moreover, we carefully develop a BRAM-only matching process to fully utilize FPGA's on-chip memory, which avoids the expensive intermediate data transfer between FPGA's BRAM and DRAM. Comprehensive experiments show that FAST achieves up to 462.0x and 150.0x speedup compared with the state-of-the-art algorithm *DAF* and *CECI*, respectively. In addition, FAST is the only algorithm that can handle the billion-scale graph using one machine in our experiments.**

***Index Terms*—subgraph matching, FPGA, pipeline**

I. Introduction

Graph analysis has been playing an increasingly important role in the area of data analytics in recent years. One of the most fundamental problems in graph analysis is subgraph matching. Given a query graph q and a data graph G, it aims to find all subgraphs of G that are isomorphic to q. It has a wide range of applications including protein-protein interaction networks analysis [27], chemical sub-compound search [38], social network analysis [32], computer aided design [26], and graph pattern mining [34]. It is also a core operation in graph databases [8] and RDF engines [43]. However, it is challenging to compute subgraph matching, especially in large graphs, due to its NP-hardness [19].

Extensive research has been conducted to develop efficient solutions for subgraph matching. Most practical solutions on CPUs [12]–[14], [17], [18], [20], [31], [40] are based on the backtracking approach, which recursively extends a partial embedding by mapping the next query vertex to a data vertex. Limited by the stand-alone design, these sequential solutions show unsatisfactory response time and poor scalability when handling massive graphs. In addition, general-purpose CPUs are not an ideal way to handle graph processing: they do not offer flexible high-degree parallelism, and their caches do not work effectively for irregular graphs with limited data locality.

FPGAs. FPGAs, which provide a new alternative to accelerate computation in the hardware level, has evolved rapidly in recent years. FPGAs have shown enormous advantages over CPUs on parallelism. Data can be directly streamed to FPGAs without instruction decoding and processed in pipelines. Because of its high potential to express parallelism at a massive scale and other benefits such as more energy-efficient than GPUs [11], FPGAs have been applied to implement complex systems in industry. For example, Microsoft used FPGAs to speed up Bing Search and Azure Machine Learning [1]. FPGAs have also been rolled out by major cloud service providers such as Amazon Web Services [2], Alibaba [3], Tencent [4], Huawei [5], and Nimbix [6]. In academia, it has become a promising trend to use FPGAs to speed up different research problems including many graph processing problems [10], [15], [25], [41], [42]. Nevertheless, subgraph matching algorithms using FPGAs have not been developed in the literature. FPGA-based subgraph matching can speed up and benefit all aforementioned applications. It can also be integrated into existing graph database systems (e.g. Neo4j [8]) and RDF engines (e.g. gStore [43]) to accelerate various subgraph queries. Motivated by this, in this paper, we explored how the pipeline mechanism of FPGAs can be fully utilized to accelerate the subgraph matching problem.

Challenges. We present the challenges of solving the problem of subgraph matching on FPGAs as follows:

- *Strictly pipelined design on FPGA*. FPGAs utilize a pipelined design, in which a fully pipelined loop demands no data dependencies among iterations. Thus the existing backtracking-based algorithms cannot be directly implemented on FPGAs. Furthermore, as FPGAs have an order of lower clock frequency than CPUs (e.g., 300MHz vs. 2GHz), it requires intricate design of the subgraph matching units on FPGAs to obtain high performance.
- *Limited FPGA on-chip memory*. FPGAs have small sizes of on-chip memory (BRAM) that are usually only tens of megabytes; hence the huge graph data and intermediate results will easily overflow BRAM when performing subgraph matching on FPGAs. Moreover, as fetching data from FPGA's external memory (DRAM) takes much more cycles than BRAM (e.g., 8 cycles vs. 1 cycle), frequent data transfer between BRAM and DRAM can significantly harm the performance. Thus, it is rather challenging to manage the data on FPGAs efficiently such that we can reduce the data transfer operations between BRAM and DRAM.

Contributions. To address these challenges, we propose a CPU-FPGA co-designed architecture which accelerates subgraph matching on a single machine using the power of FPGAs. Specifically, our main contributions are as follows.

- *The first CPU-FPGA co-designed framework to accelerate subgraph matching.* The framework includes a well-designed scheduler on the host side (i.e., the CPU) and a fully pipelined matching algorithm FAST on the kernel side (i.e., the FPGA). A workload estimation method is proposed on the host side for load-balancing between the CPU and FPGA, which can be exploited to extend our framework to multi-FPGA environment. To further improve the efficiency of FAST algorithm on the kernel side, we propose two optimizations with task parallelism and task generator separation.
- *A BRAM-only matching process to fully utilize FPGA's on-chip memory.* We first design an auxiliary data structure CST to serve as a complete search space. An efficient partition strategy of CST is proposed so that CST can be fully loaded into BRAM, reducing the costly data fetching from FPGA's external memory. Then we propose a BRAM-only partial results buffer to avoid the expensive intermediate data transfer between BRAM and DRAM.
- *Extensive experiments using the industrial-standard LDBC benchmark.* Our experiments using LDBC [7] show that FAST outperforms the state-of-the-art algorithms by orders of magnitude (up to 150.0x and 462.0x compared with *CECI* [12] and *DAF* [17], respectively). More importantly, FAST is the only algorithm that can scale to the billion-scale graph on a single machine in our experiment.

Paper Organization. The rest of the paper is organized as follows. Section II introduces background and Section III presents related works. The system overview of the proposed solution is introduced in Section IV, followed by the detailed design of software and hardware in Section V and Section VI, respectively. Experimental results are presented in Section VII. Section VIII concludes the paper.

II. Background

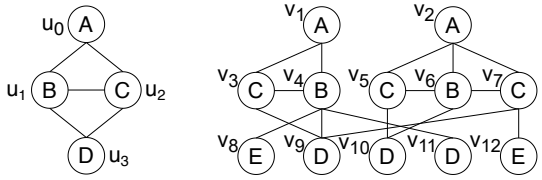
In this section, the problem definition of subgraph matching is stated first, followed by a brief introduction of FPGAs.

A. Problem Definition

A graph G is represented as a tuple $G = (V, E, l, \Sigma)$, where $V(G)$ is the set of vertices, $E(G) \subset V \times V$ is the set of edges in G, Σ is the set of labels, and l is a labelling function that assigns each vertex $v \in V$ a label in Σ, denoted $l_G(v)$. We focus on *undirected*, *labelled*, *connected*, and *simple* graphs in this paper. Note that, our techniques can be readily extended to edge-labeled and directed graphs. We denote the number of vertices and edges in G by $|V(G)|$ and $|E(G)|$, respectively. The set of neighbors of $v \in V(G)$ in G is denoted by $N_G(v) = \{v' \in V(G) | (v, v') \in E(G)\}$ and the degree of v, denoted by $d_G(v)$, that is $d_G(v) = |N_G(v)|$. The $\overline{d}_G = \frac{2|E(G)|}{|V(G)|}$ and D_G are denoted as the average and maximum degree, respectively.

Definition 1. (Subgraph Isomorphism) Given a query graph q and a data graph G, q is subgraph isomorphism to G if and only if there is an *injective* mapping M from $V(q)$ to $V(G)$ such that $\forall u \in V(q)$, $l_q(u) = l_G(M(u))$ and $\forall (u, u') \in E(q)$, $(M(u), M(u')) \in E(G)$, where $M(u)$ is the vertex to which u is mapped.

We refer to each injective mapping M as a *subgraph isomorphism embedding* of q in G. A graph g' is an *induced subgraph* of g if and only if $\forall \mu, \mu' \in V_{g'}, e = (\mu, \mu') \in E_g$, we have $e \in E_{g'}$. We call an embedding of an induced subgraph of q in G a *partial embedding*, denoted as p. The $M_p(u)$ denotes the mapping vertex of u in q. We use $\mathcal{O}$ to denote the *matching order*, which is a sequence of query vertices representing the order they are matched.



(a) Query graph (b) Data graph

Fig. 1. Subgraph Matching

Example 1. For example, consider the query graph q in Fig. 1(a) and the data graph G in Fig. 1(b). Suppose the matching order is $\{u_0, u_1, u_2, u_3\}$, since there is a subgraph isomorphism embedding $M = \{(u_0, v_1), (u_1, v_4), (u_2, v_3), (u_3, v_9)\}$, q is subgraph isomorphism to G. We call $p = \{(u_0, v_1), (u_1, v_4), (u_2, v_3)\}$ a partial embedding and u_3 will be the next query vertex to match.

Problem Statement. Given a query graph q and a data graph G, we study the problem of subgraph matching, which efficiently extracts all subgraph isomorphic embeddings of q in G.

B. Characteristics of FPGA

A field-programmable gate array (FPGA) is an integrated circuits that consists of a matrix of configurable logic, memory, and digital signal processing (DSP) components. These components are distributed within a grid of configurable routing wires connected to programmable chip I/O blocks. This flexible and programmable fabric can be configured to perform any functionality implemented as a digital circuit. The FPGA program statements are translated into a netlist of primitive components first and then be assigned to physical components in the FPGA fabric, determining which routing wires should be used to connect them. This architecture allows data to be directly streamed to FPGAs with no need to decode instructions, as necessary in CPUs, to achieve high efficiency.

FPGAs have an unique programming model in which computations are laid out spatially and the programmer has to specify how data and control flows from one logic block to another inside the data path. Thus, common design challenges when developing FPGA-based algorithms is the amount of space (resources) required and the ability to meet timing (ensuring the data can be moved across the circuit in a correct manner). It is also worth to mention that the clock rate on FPGAs is usually about 10x slower than that of CPUs (i.e.,

300MHz vs 2.4GHz). Thus, FPGA-based algorithms must be thoughtfully designed to provide better performance than CPU implementations, by exploiting massive parallelism, typically in the form of deep pipelines.

III. Related Work

A. Subgraph Matching

Stand-alone Solutions. The study of practical subgraph matching algorithms was initiated by Ullmann's backtracking algorithm [36], which recursively matches query vertices to data vertices following a given matching order. Later researches [14], [20], [31], [40] focus on different matching order, pruning rules, and index structure. Turbo$_{ISO}$ [18] proposes to merge similar vertices with and a *CR* index structure. CFL-Match [13] proposes the core-forest-leaf decomposition to reduce redundant Cartesian products and proposes a more compact auxiliary structure *CPI* to solve the exponential size of *CR*. CECI [12] and DAF [17] adopt the intersection-based method to find the candidates, which demonstrate better performance than the edge verification method used in previous works [33]. However, these solutions fail to accommodate large graphs due to their inherent sequential nature.

Distributed Solutions. Most distributed algorithms utilize distributed join to compute matches. [22], [23], [28] decompose the query graph into sub-queries, find the matches of each subquery, and use a series of binary joins to assemble the final results. [9], on the other hand, grows the query graph one vertex at a time following a specific order to obtain worst-case optimality. FAST can be potentially used to accelerate the computation in distributed subgraph matching.

GPU-based Solutions. GpSM [35] and GunrockSM [37] adopt the binary join strategy in GPUs, which collects candidates for each edge of q and joining them to find final matches. They suffer from high computation workload, high memory latency, and severe workload imbalance. GSI [39] proposes a Prealloc-Combine approach, which joins candidate vertices instead of edges to improve the efficiency. The algorithms mentioned above are only able to handle the graphs that can be fit into the GPU memory. PBE [16] solves this by partitioning the graph in advance and matching intra- and inter-partition matches in two separate steps. However, as the on-chip memory of an FPGA is order-of-magnitude smaller than a GPU memory, this approach can hardly be applied to FPGAs.

B. FPGA-based Acceleration of Graph Processing

FPGAs can be an energy-efficient solution to deliver specialized hardware for graph processing. This is reflected by the recent interests in developing various graph algorithms and graph processing frameworks on FPGAs. For examples, [10] applies FPGAs to speed up *Maximum Matching* and [42] utilizes FPGAs to accelerate the process of the *Single-Source-Shortest-Paths*. In addition to these specific graph algorithms on FPGAs, a lot of effort was devoted to design generic frameworks for facilitating the implementation of graph algorithms on FPGAs [15], [25], [41]. However, these frameworks are usually built upon specific programming models (e.g. BSP, Vertex-Centric) supporting only limited APIs. This restricts the implementation of a highly optimized subgraph matching algorithm. More critically, most of the frameworks can only handle small graphs and cannot scale to large ones.

IV. System Overview

The overview architecture of our system is illustrated in Fig. 2. The host side, i.e. CPU, takes charge of constructing and partitioning our novel auxiliary data structure CST and offloading them to FPGA through PCIe bus. It also shares a small portion of matching tasks to improve throughput. The kernel side, i.e. FPGA card[1], is PCIe-attached to the host machine, focusing on the subgraph matching tasks.

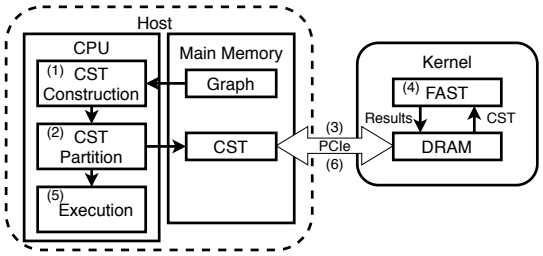


Fig. 2. The overall system architecture

When the query and data graph are read into the host's main memory, the system launches the execution tasks described as follows:

1) CPU constructs CST based on q and G, which prunes a large number of false positives according to graph attributes such as labels and degrees, etc. CST serves as a complete search space for all embeddings of q in G (Section V-A).
2) Limited by FPGA on-chip resources, CST is often too large to be fully loaded into BRAM. The host side partitions CST to satisfy the size constraint (Section V-B).
3) Once a partitioned CST satisfies the constraint, it is transferred to DRAM on FPGA card from the host's main memory through PCIe bus.
4) On the kernel side, FAST reads a partitioned CST from DRAM to BRAM and runs subgraph matching on it. The results are flushed to DRAM when the whole search space of this CST has been searched. FAST repeats this procedure as long as there exists an unprocessed CST (Section VI).
5) On the host side, when all CST has been partitioned and offloaded, CPU shares a small portion of matching tasks to improve the overall throughput (Section V-C).
6) When FPGA finishes its processing, CPU receives a termination signal and fetches results to the main memory.

The details of software and hardware implementation are described in Section V and Section VI, respectively.

V. Software Implementation

In this section, we first introduce our novel auxiliary data structure CST and its partition strategy. Then we present how to schedule matching tasks between the host and kernel side.

[1]In this paper, we focus on FPGAs with DRAM attached, while our techniques can be applied on FPGAs without DRAM as well.

TABLE I
DEFINITION OF PARAMETERS IN SOFTWARE IMPLEMENTATION

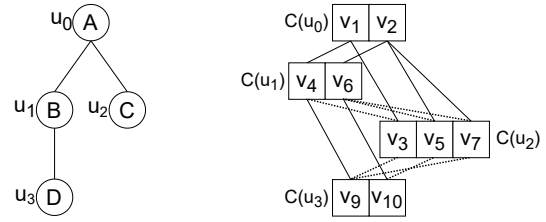
Symbol	Definition
CST	candidate search tree
t_q	a breadth-first search tree of q
$C(u)$	the candidate set of u in CST
$N^u_{u'}(v)$	the adjacency list of v regarding (u, u')
u_p / u_c	the parent / child vertex of u in CST
u_n	the non-tree neighbor of u in CST
$\mathcal{O}$	the matching order of q
$\|$CST$\|$	the size of CST
D_{CST}	the maximum degree of candidates in CST

A. CST *Structure*

We adopt the indexing-enumeration framework; that is, construct an auxiliary data structure, then compute all embeddings based on this data structure. Following conventional technique [13], [30], the query graph is firstly transformed into a spanning tree. Given a query graph q and a data graph G, we build an auxiliary data structure upon them called *candidate search tree* (CST).

Definition 2. (Candidate Search Tree) Given a query graph q and a data graph G, a candidate search tree $\text{CST}_{(q,G)}$ is a graph[2] that is isomorphic to q. Each vertex u of $\text{CST}_{(q,G)}$ has a candidate set, denoted $C(u)$, which stores all vertices of G that u can be mapped. There is an edge between $v \in C(u)$ and $v' \in C(u')$ for adjacent vertices u and u' in $\text{CST}_{(q,G)}$ if and only if $(v, v') \in E(G)$.

We denote $\text{CST}_{(q,G)}$ as CST if the context is clear. Given the query graph q and its BFS trees t_q, we call adjacent vertices u and u_n in CST *non-tree neighbors* if $(u, u_n) \in E(q)$ but $(u, u_n) \notin E(t_q)$. The adjacent candidates $v \in C(u)$ and $v_n \in C(u_n)$ for non-tree neighbors u and u_n in CST are called *non-tree candidate neighbors*. We use $N^u_{u'}(v)$ to denote the adjacency list of $v \in C(u)$ with respect to (u, u') in CST, i.e., $N^u_{u'}(v) = \{v' \in C(u') \mid (v, v') \in E(\text{CST})\}$. CST inherits the parent-child relationships of t_q. We use u_p and u_c to denote the parent and child vertex of u, respectively. The vertex u in CST is a *leaf* or *root* vertex if u has no child vertices or parent vertices, respectively.



(a) BFS Tree t_q (b) CST

Fig. 3. Example CST structure

Example 2. For example, given the query graph q, the data graph G in Fig. 1 and BFS tree t_q of q in Fig. 3(a), the corresponding CST is in Fig. 3(b). Then u_1 and u_2 are called non-tree neighbors because $(u_1, u_2) \notin E(t_q)$,while $v_4 \in C(u_1)$ and $v_3 \in C(u_2)$ are called non-tree candidate neighbors. $C(u_1) = \{v_4, v_6\}$, $C(u_2) = \{v_3, v_5, v_7\}$, $N^{u_1}_{u_2}(v_6) = \{v_5, v_7\}$ and $N^{u_2}_{u_3}(v_3) = \{v_9\}$. The all embeddings of q in G $\{(u_0, v_1), (u_1, v_4), (u_2, v_3), (u_3, v_9)\}$ and $\{(u_0, v_2), (u_1, v_6), (u_2, v_5), (u_3, v_{10})\}$ can be computed by traversing only the CST.

[2]We abuse the term *tree* during naming to emphasize that CST is constructed based on the spanning tree of the query.

Algorithm 1: CSTConstructor(q, G, t_q)

Input: q, G, t_q
Output: CST
1 $root \leftarrow$ root vertex of t_q;
2 $C(root) \leftarrow$ compute candidates of $root$;
/* Line 3-7: Top-Down Construction */
3 **foreach** $u \in V(q)$ *in a top-down fashion* **do**
4 $C(u) \leftarrow$ compute candidates of u;
5 **foreach** $v_p \in C(u_p)$ **do**
6 **foreach** $v \in C(u)$ **do**
7 **if** $(v, v_p) \in E(G)$ **then** $N^{u_p}_u(v_p).push(v)$;
/* Line 8-14: Bottom-Up Refinement */
8 **foreach** $u \in V(q)$ *in a bottom-up fashion* **do**
9 **foreach** $v \in C(u)$ **do**
10 **if** v *is not valid* **then**
11 remove v and its adjacency lists;
12 **foreach** *child vertex* u_c *of* u *in* t_q **do**
13 **foreach** $v' \in N^u_{u_c}(v)$ **do**
14 **if** $v' \notin C(u_c)$ **then** remove v' from $N^u_{u_c}(v)$;
/* Line 15-19: Add Edges Between Non-tree Candidate Neighbors */
15 **foreach** $u \in V(q)$ **do**
16 **foreach** $v \in C(u)$ **do**
17 **foreach** *non-tree neighbor* u_n *of* u **do**
18 **foreach** $v_n \in C(u_n)$ **do**
19 **if** $(v, v_n) \in E(G)$ **then** $N^u_{u_n}(v).push(v_n)$;
20 **return** CST

The construction of CST is described in Alogrithm 1. We first adopt the similar top-down construction (Line 3-7) and bottom-up refinement (Line 8-14) in [13] to build a tree-like data structure. We verify whether a data vertex conforms with the local features of the query vertex to compute candidate set $C(u)$ (Line 2, Line 4). A candidate v of vertex u is valid if $|N^u_{u_c}(v)| \neq 0$ for any child vertex u_c of u and $\exists v_p \in C(u_p)$ that $v \in N^{u_p}_u(v_p)$. We remove v from $C(u)$ and its adjacency lists if v is not valid during the bottom-up refinement. (Line 10-11). Then edges are added between non-tree candidate neighbors (Line 15-19).

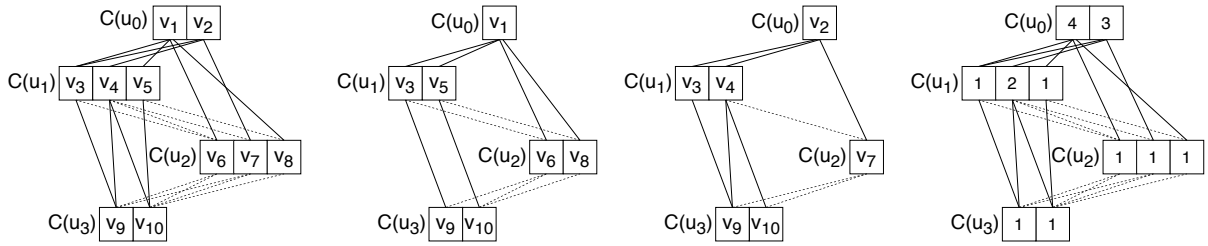
Soundness. CST should serve as a complete search space for the given query graph q over the data graph G. To achieve this, CST must satisfy the following soundness constraint:

- *For every vertex u in* CST*, if there is an embedding of q in G that maps u to v, then v must be in $C(u)$.*

Note that, although in the soundness requirement we only consider candidates of query vertices, the edges between candidates are automatically included based on our CST definition. Regarding a sound CST, we have the following theorem.

Theorem 1. Given a sound CST, all embeddings of q in G can be computed by traversing only the CST.

Remark. CST has vital differences with the auxiliary data structure in previous works, namely *CPI (compact path-index)* [13] and *CS (candidate space)* [17]. Compared with *CPI*, CST uses all edge information in q during construction (by adding



(a) Initial CST with $k = 2$ (b) 1st partition of CST (c) 2nd partition of CST (d) Workload Estimation

Fig. 4. Running Example of Scheduling

non-tree edges), making it a complete search space. Hence, it can be partitioned and the embeddings of each partition can be computed independently in FPGA's BRAM (details will be introduced in the next subsection). The reasons that we do not use the structure *CS* are as follows: (1) The top-down construction and bottom-up refinement of CST is equivalent to the first two refinements (totally three) of *CS*, making the size of CST close to *CS* for most data graphs; (2) Constructing CST is much less expensive because the edges between non-tree candidate neighbors are not updated during construction as necessary in *CS*. Consequently, CST can potentially have a larger search space than *CS* because of fewer pruning steps. However, there is an essential trade-off between the size of search space and the construction cost. Compared with pure CPU-based algorithms, FAST is more sensitive to the cost of constructing the auxiliary data structure conducted by CPU, to let FPGA receive its tasks from the host as soon as possible.

B. CST Partition

Limited by on-chip resources on FPGAs, CST is often too large to be fully loaded into BRAM. Generally, the read latency of BRAM is 1 cycle while DRAM is about 7-8 cycles. Our experiments show the dramatic performance decreasing when we access CST from DRAM rather than BRAM (Section VII-A). On the other hand, accesses to CST are random and unpredictable, which eliminates the possibility of prefetching the data from DRAM to BRAM. Hence, it is necessary to partition CST and offload them to FPGA one by one.

In addition to the size of CST, denoted as $|CST|$, we also set a limitation on the maximum degree of candidates in CST, i.e., D_{CST}. The reason is that the maximum number of access ports to an adjacency list are limited on FPGAs and it will be discussed in detail in Section VI-A. We use δ_S and δ_D to denote the threshold of $|\mathsf{CST}|$ and D_{CST}, respectively. We partition the CST if either $|CST| > \delta_S$ or $D_{CST} > \delta_D$.

The partition strategy of CST is illustrated in Algorithm 2. Note that we adopt the path-based method to compute the matching order $\mathcal{O}$ in this paper, which determines $\mathcal{O}$ by ordering the root-to-leaf paths of t_q. However, our method is designed to work with any arbitrary connected matching orders. Initially, to partition CST, we partition candidates of root vertex in CST. If there is only one candidate of root vertex in CST, we move on to partition candidates of next vertex u in $\mathcal{O}$. The first step is to determine the partition factor k, which equals to the maximum value between the ratio of $|CST|$ and D_{CST} to their corresponding thresholds (Line 2). If k exceeds the number of candidates, i.e., $|C(u)|$, we set k to $|C(u)|$ (Line 3). We partition $C(u)$ into k parts evenly and then construct a new CST level-by-level in a top-down manner. For those vertices precedes u in $\mathcal{O}$, we pick candidates as the same as the old CST (Line 7-8). For those vertices follows u in $\mathcal{O}$, we pick candidates in old CST which can reach at least one candidate in the partitioned $C(u)$ (Line 9-12). CST is offloaded to FPGA or assigned to CPU as soon as it satisfies $|CST|$ and D_{CST} constraints (Line 15). Otherwise, it will be further partitioned recursively.

Algorithm 2: CSTPartition(CST,$\mathcal{O}, index$)

```
   Input: CST,O, index
1  u ← O[index];
2  k ← max(|CST|/δ_S, D_CST/δ_D);
3  k ← min(k, CST.|C(u)|);
4  partition CST.C(u) into k parts evenly;
5  for i from 0 to k do
6      CST' ← ∅;
7      foreach vertex u' precedes u in O do
8          CST'.C(u') ← CST.C(u');
9      foreach vertex u' follows u in O do
10         foreach candidate v in CST.C(u') do
11             if v can reach ith partitioned CST.C(u) then
12                 CST'.C(u') ← CST'.C(u') ∪ {v};
13     update adjacency lists of CST' based on CST;
14     if |CST'| ≤ δ_S and D_CST' ≤ δ_D then
15         CSTProcess(O, CST');
16     else if CST'.|C(u)| equals to 1 then
17         CSTPartition(CST', O, index + 1);
18     else CSTPartition(CST', O, index);
```

Example 3. As shown in Fig. 4(a), suppose k is 2, we first partition the root candidates $\{v_1, v_2\}$ into 2 parts: $\{v_1\}$ and $\{v_2\}$. Then, to construct CST rooted by v_1, we pick candidates of u_1 and u_2 that are adjacent to v_1, which are $\{v_3, v_5\}$ and $\{v_6, v_8\}$. After that, we pick the candidates of u_3 that can reach v_1, which are $\{v_9, v_{10}\}$. Obviously, there is no overlap of the search space between two partitioned CST in Fig. 4(b) and Fig. 4(c), so no repeated results will be reported.

C. Schedule the Matching Tasks

After finishing the partition of CST, the host side shares a small portion of matching tasks in order to further improve the throughput as a whole. Considering load balancing between the CPU and FPGA, the workload of CST, denoted as W_{CST}, should be estimated first. The size of search space in different CST can usually differs a lot due to the power-law feature of real-world graphs. We use the number of embeddings in

CST without considering any false positives to estimate W_{CST}. It can be computed in a bottom-up way using a dynamic programming algorithm. For each candidate $v \in C(u)$, we compute $c_u(v)$, the number of embeddings in CST for the subgraph of q induced by the suffix of matching order starting from u such that u is mapped to v. Initially, $c_u(v) = 1$ for all leaf vertices u. Then we compute $c_u(v)$ in a bottom-up fashion, where $c_u(v) = \prod_{u' \in u.child} \sum_{v' \in N^u_{u'}(v)} c_{u'}(v')$. Finally, the total workload $W_{\mathsf{CST}} = \sum_{v \in C(u_r)} c_{u_r}(v)$.

Example 4. Given CST in Fig. 4(a) and t_q in Fig. 3(a), the workload estimation results are illustrated in Fig. 4(d). For leaf vertices u_3 and $u2$, $c_{u_3}(v_9) = c_{u_3}(v_{10}) = c_{u_2}(v_6) = c_{u_2}(v_7) = c_{u_2}(v_8) = 1$. Then we compute $c_u(v)$ in a bottom-up fashion, e.g., $c_{u_0}(v_1) = (c_{u_1}(v_3) + c_{u_1}(v_5)) * (c_{u_2}(v_6) + c_{u_2}(v_8)) = 4$. Finally, $W_{\mathsf{CST}} = c_{u_0}(v_1) + c_{u_0}(v_2) = 4 + 3 = 7$.

Algorithm 3: CSTProcess($\mathcal{O}$, CST)

Input: $\mathcal{O}$, CST
1 $W_{\mathsf{CST}} \leftarrow$compute the workload of CST;
2 **if** $W_C + W_{CST} < \delta \times (W_C + W_F + W_{\mathsf{CST}})$ **then**
3 assign CST to CPU;
4 $W_C \leftarrow W_C + W_{CST}$;
5 **else**
6 FAST(CST, $\mathcal{O}$);
7 $W_F \leftarrow W_F + W_{CST}$;

As illustrated in Algorithm 3, we restrict the proportion of the total workload of CST assigned to the host side from exceeding a threshold, denoted as δ. When a valid CST is constructed, W_{CST} is computed first (Line 1). We use W_C and W_F to denote the total workload of CST assigned to the host and kernel side, respectively. If $\frac{W_C + W_{\mathsf{CST}}}{W_C + W_F + W_{\mathsf{CST}}} < \delta$, it will be assigned to the host side (Line 3-4). Otherwise, FPGA takes charge of this CST (Line 3-4). The host side uses the basic backtracking subgraph matching algorithm to process CST. It should be noted that when CST is assigned to CPU, CST is temporarily cached and will be processed when all partition procedure finishes. When CST is assigned to FPGA, CST is offloaded to FPGA immediately.

VI. Hardware Implementation

In this section, we first present our proposed algorithm FAST to accelerate subgraph matching on FPGAs. Then we introduce several important optimizations to improve the matching process based on FPGA characteristics. Notations of all the related parameters are listed in Table II.

TABLE II
DEFINITION OF PARAMETERS

Symbol	Definition
$\mathcal{P}$	Intermediate results buffer
$\mathcal{M}$	the set of all embeddings
p_i / p_o	An input / output partial result
$\mathcal{P}_o$	The set of p_o
N_o	The maximum size of $\mathcal{P}_o$
u_p / u_n	The parent / non-tree neighbor of u in CST
t_v / t_n	a visited / edge validation task
$\mathcal{T}_v$ / $\mathcal{T}_n$	The set of t_v / t_n

A. Basic Pipeline of Subgraph Matching

In the typical backtracking algorithms [12], [13], [17], [18], one partial result is expanded at a time by matching the next vertex to a candidate vertex following the matching order. This sequential design cannot be pipelined because of data dependencies among iterations. To solve this, we decompose the matching process into three steps as follows: (1) *Generator* expands partial results by matching the next vertex in the matching order; (2) *Validator* verifies whether a new partial result is valid; (3) *Synchronizer* collects results. Different from the typical algorithms, our method processes thousands of partial results at a time in these steps, so that each step can fully utilize the pipeline mechanism of FPGA. Our basic pipeline design is shown in Algorithm 4, denoted as **FAST**.

Algorithm 4: FAST(CST, $\mathcal{O}$)

Input: CST, $\mathcal{O}$
Output: $\mathcal{M}$
1 $\mathcal{M} \leftarrow \emptyset$; $\mathcal{P} \leftarrow \emptyset$;
2 **foreach** *candidate v of root vertex* ***pipeline*** **do**
3 $\mathcal{P}.push(\{v\})$;
4 **while** $\mathcal{P} \neq \emptyset$ **do**
5 $\mathcal{P}_o, \mathcal{T}_v, \mathcal{T}_n \leftarrow Generator(\mathcal{P}, \mathsf{CST}, \mathcal{O})$;
6 $\mathcal{B}_v \leftarrow VisitedValidator(\mathcal{T}_v)$;
7 $\mathcal{B}_n \leftarrow EdgeValidator(\mathsf{CST}, \mathcal{T}_n)$;
8 $Synchronizer(\mathcal{M}, \mathcal{P}, \mathcal{P}_o, \mathcal{B}_v, \mathcal{B}_n)$;
9 **return** $\mathcal{M}$

Given CST and matching order $\mathcal{O}$, we first match the root vertex to all its candidates to generate first batch of partial results (Line 2-3). Then for each round, *Generator* reads multiple partial results from $\mathcal{P}$ and expand them (Line 5). A partial result is valid iff it passes the two validations: (1) *visited validation*, i.e., the new mapped candidate v is not visited before (Line 6); (2) *edge validation*, i.e., the new mapped candidate v are adjacent to the mapping vertices of u's non-tree neighbors (Line 7). The new valid partial or complete results will be pushed into $\mathcal{P}$ or $\mathcal{M}$ by *Synchronizer*, respectively (Line 8). **FAST** terminates when $\mathcal{P}$ is empty (Line 4). As shown in Fig. 5(a), these steps are processed serially in our basic pipeline design. We discuss the details as follows.

Algorithm 5: Generator($\mathcal{P}$, CST, $\mathcal{O}$)

Input: $\mathcal{P}$, CST, $\mathcal{O}$
Output: $\mathcal{P}_o, \mathcal{T}_v, \mathcal{T}_n$
1 $\mathcal{P}_o \leftarrow \emptyset$; $\mathcal{T}_v \leftarrow \emptyset$; $\mathcal{T}_n \leftarrow \emptyset$;
2 $u \leftarrow$ *get next vertex to be mapped in* $\mathcal{O}$;
  `/* Line 3-9: Generate` $\mathcal{P}_o$ `and` $\mathcal{T}_v$ `*/`
3 **while** $|\mathcal{P}_o| < N_o$ **do**
4 $p_i \leftarrow \mathcal{P}.pop()$;
5 $C(u) \leftarrow$ *get u's candidates from* CST *based on* p_i;
6 **if** $|\mathcal{P}_o| + |C(u)| > N_o$ **then** break;
7 **foreach** $v \in C(u)$ ***pipeline*** **do**
8 $\mathcal{P}_o.push(p_i \times \{v\})$;
9 $\mathcal{T}_v.push((v,\ p_i))$;
  `/* Line 10-12: Generate` $\mathcal{T}_n$ `*/`
10 **foreach** *u's non-tree neighbor u_n* **do**
11 **foreach** $p_o \in \mathcal{P}_o$ ***pipeline*** **do**
12 $\mathcal{T}_n.push(M_{p_o}(u),\ M_{p_o}(u_n),$ *the index of* $p_o)$;
13 **return** $\mathcal{P}_o, \mathcal{T}_v, \mathcal{T}_n$

Generator. *Generator* is used to expand partial results and generate *visited validation* tasks $\mathcal{T}_v$ and *edge validation* tasks $\mathcal{T}_n$. Algorithm 5 shows the workflow of *Generator*. At first,

we expand partial results and generate visited validation tasks $\mathcal{T}_v$ (Line 3-9). This procedure can be fully pipelined. Limited by on-chip resources, we control the maximum number of newly expanded partial results each round, denoted as N_o (line 6). We will discuss how to pick the value of N_o in detail in Section VI-B. Then we generate edge validation tasks $\mathcal{T}_n$ (Line 10-12). The inner loop of t_n generation procedure is fully pipelined (Line 11-12). We have specific one visited validation task t_v for each new partial result p_o, while the number of edge validation tasks t_n is determined by the query structure and matching order. One precondition to pipeline a loop is that the cycles of loop body are fixed. So we have to separate $\mathcal{T}_n$ generation procedure from other two steps. The outer loop of $\mathcal{T}_n$ generation procedure (Line 10) cannot be pipelined for the same reason.

Algorithm 6: VisitedValidator($\mathcal{T}_v$)

```
   Input: T_v
   Output: B_v
1  B_v ← ∅;
2  foreach (v, p_i) in T_v pipeline do
3  |   b ← 1;
4  |   foreach v' in p_i parallel do
5  |   |   if v' == v then b ← b & 0;
6  |   B_v.push(b);
7  return B_v
```

Visited Validator. As shown in Algorithm 6, *Visited Validator* is used to validate if the new mapped candidate v is visited before by comparing v with every vertex in p_i (Line 4-5). We use the array partition mechanism in FPGAs, i.e., partitioning an array into individual elements, to effectively increases the amount of read and write ports for the storage. The mechanism offers the possibility to compare v with every element of p_o in parallel. Each p_o has two bits to reflect whether it passes visited and edge validation, respectively. If v has been visited, the visited bit is set to zero (Line 6). This module can be pipelined completely.

Algorithm 7: EdgeValidator($\mathsf{CST}, \mathcal{T}_n$)

```
   Input: CST, T_n
   Output: B_n
1  B_n ← ∅;
2  for (v, v_n, i) in T_n pipeline do
3  |   if (v, v_n) exists in CST then b ← 1;
4  |   else b ← 0 ;
5  |   B_n.set(i, b);
6  return B_n
```

Edge Validator. As shown in Algorithm 7, *Edge Validator* checks whether the new mapped candidate v is adjacent to all v_n, the mappings of u's non-tree neighbors. It checks edge existence in CST (Line 3) by comparing v_n with all non-tree candidate neighbors of v. Here we also adopts the array partition mechanism so that edge existence check can be completed in $O(1)$. However, this mechanism costs much more on-chip resources, which limits the maximum number of access ports of an array, denoted as $Port_{max}$. Thus we partition CST if D_{CST} exceeds $Port_{max}$. If there is no edge between v and v_n, the edge bit is set to zero (Line 4). It should be noted that each p_o may have more than one t_n, any of them failed will lead to an invalid p_o. The *Edge Validator* module can also been pipelined completely.

Algorithm 8: Synchronizer($\mathcal{M}, \mathcal{P}, \mathcal{P}_o, \mathcal{B}_v, \mathcal{B}_n$)

```
   Input: M, P, P_o, B_v, B_n
1  for p_o in P_o pipeline do
2  |   b_v ← B_v.pop(), b_n ← B_n.pop();
3  |   if b_v = 1 and b_n = 1 then
4  |   |   if |p_o| == |O| then M.push(p_o) ;
5  |   |   else P.push(p_o) ;
```

Synchronizer. As shown in Algorithm 8, *Synchronizer* is designed to collect partial results. For each p_o, it first fetches its two validation bits from $\mathcal{B}_v$ and $\mathcal{B}_n$ (Line 2). If any bit is zero, this p_o will be discarded (Line 3). Then it compares $|p_o|$ and $|\mathcal{O}|$ to check whether it is a complete result (Line 4). The complete result is reported and stored into $\mathcal{M}$ while the partial result is stored back into $\mathcal{P}$.

Example 5. Suppose that we have CST in Fig. 4(b), $\mathcal{O} = (u_0, u_1, u_2, u_3)$ and $\mathcal{P} = \{\{v_1, v_3\}, \{v_1, v_5\}\}$. The *Generator* first expand partial results in $\mathcal{P}$ to get $\mathcal{P}_o = \{\{v_1, v_3, v_6\}, \{v_1, v_3, v_8\}, \{v_1, v_5, v_6\}, \{v_1, v_5, v_8\}\}$ and generate $\mathcal{T}_v = \{(v_6, 0), (v_8, 0), (v_6, 1), (v_8, 1)\}$. After that, $\mathcal{T}_n = \{(v_3, v_6, 0), (v_3, v_8, 1), (v_5, v_6, 2), (v_5, v_8, 3)\}$ are generated. Then *Visited Validator* and *Edge Validator* processes $\mathcal{T}_v$ and $\mathcal{T}_n$, respectively. We get $\mathcal{B}_v = \{1, 1, 1, 1\}$ and $\mathcal{B}_n = \{1, 0, 0, 1\}$. Finally, *Synchronizer* pushes valid partial results $\{\{v_1, v_3, v_6\}, \{v_1, v_5, v_8\}\}$ into $\mathcal{P}$.

B. Cycle Analysis and Buffer Design

In this subsection, we first discuss how to pick the value of the maximum number of newly expanded partial results each round, denoted as N_o. Then our BRAM-only intermediate results buffer is introduced, which completely avoids the intermediate data transfer between BRAM and DRAM.

Based on Algorithm 5-8, we use $L_1 - L_6$ to denote the average cycles for the following six procedures: (1) read from intermediate results buffer $\mathcal{P}$; (2) generate a new partial result p_o and its visited validation task t_v; (3) process t_v; (4) collect p_o; (5) generate an edge validation task t_n; (6) process t_n. We use m to denote the number of t_n for p_o. So the total cycles of a partial result from being expanded to finally collected are $(L_1 + L_2 + L_3 + L_4 + n \times (L_5 + L_6))$.

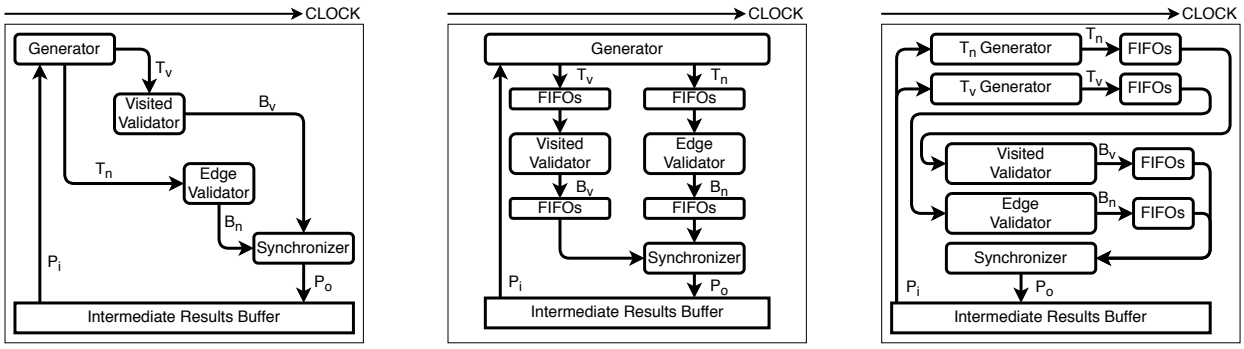
Suppose in the whole search space, the total number of p_o and t_n is N and M, respectively. To simplify the equations, we denote $\sum_{j=1}^{4} L_i$ as L_f and $\sum_{j=5}^{6} L_i$ as L_t. So without any pipelining optimization, the total cycles L_{serial} to process the whole search space is:

$$L_{serial} = N \times L_f + M \times L_t \tag{1}$$

In FAST, the six procedures can be pipelined completely and we process N_o partial results each round. It means each round the serial algorithm needs $L_2 \times N_o$ cycles to process the second procedure while FAST needs $(L_2 + N_o + 1)$ cycles. So the total cycles L_{basic} is:

$$L_{basic} \approx \frac{N \times L_f + M \times L_t}{N_o} + 4N + 2M \tag{2}$$

As shown in Equation 2, a small N_o decreases the performance. However, it leads to over-consumption of on-chip resources when N_o is too large. Thus we ensure $N_o >>$



(a) Basic pipeline of subgraph matching (b) Optimization with task parallelism (c) Optimization for *Generator*

Fig. 5. The Hardware Implementation of FAST

$\frac{N\times L_f+M\times L_t}{4N+2M}$ and the specific value of N_o should be carefully chosen based on different FPGAs (our configuration is given in Section VII). It should be noted that for a partial result, if its candidates are too many, i.e., $|C(u)| > N_o$, we will generate N_o partial results by mapping N_o candidates in $C(u)$. The rest candidates will be mapped later.

It is expensive to transfer partial results between BRAM and DRAM. So we develop a strategy to avoid the overflow of the intermediate results buffer $\mathcal{P}$. We use p^n to denote a partial result that maps n query vertices. We observe that a $p^{|V(q)|}$ is a complete result and will not be pushed back into $\mathcal{P}$. Therefore, each round we expand p^n with the maximum n in $\mathcal{P}$so that these partial results can be expanded to complete ones as soon as possible. As a result, for any $n \in [1, |V(q)| - 1]$, our strategy guarantees the number of p^n does not exceed N_o. Finally, we allocate $(|V(q)|-1)\times N_o$ space for $\mathcal{P}$ on BRAM, which prevents the overflow of $\mathcal{P}$.

C. Optimization with Task Parallelism

As shown in Fig. 5(a), in our basic pipeline design, modules are executed in serial. The number of access ports to ordinary memory area on BRAM is limited, so two modules can not access the same memory simultaneously. As a result, *Visited Validator* and *Edge Validator* cannot start until all t_v and t_n are generated. *Synchronizer* will be idle before all validation tasks are finished. Therefore, as illustrated in Fig. 5(b), we utilize *task parallelism* mechanism on FPGA to allow modules being executed in parallel.

In contrast to loop parallelism, when task parallelism is deployed, different execution modules are allowed to operate simultaneously. The task parallelism is achieved by taking advantage of extra buffering introduced between the modules. The buffer is implemented by FIFOs (First in, First out) on FPGA. The output of each module will be streamed into the buffer, and the next module processes the data as long as the buffer is not empty.

As shown in Algorithm 5, once t_v is generated (Line 9), it is streamed to the FIFOs, and *Visited Validator* starts to work. Similarly, once t_n is generated (Line 12), it is streamed to the FIFOs, and *Edge Validator* starts its process. The p_o will be collected by *Synchronizer* as soon as its two validation bits are ready. Compared with the basic version, more than one module can work simultaneously.

Consider the total cycles of this task parallelism version, denoted as L_{task}. In this optimized design, the first loop of *Generator* (Line 3-9 in Algorithm 5) and *Visited Validator* (Algorithm 6) execute in parallel. And the second loop of *Generator* (Line 10-12 in Algorithm 5), *Edge Validator* (Algorithm 7) and *Synchronizer* (Algorithm 8) execute concurrently. To simplify the equation, suppose we have pick an appropriate N_o. Then we have:

$$L_{task} \approx 2N + \max(N, M) \tag{3}$$

Compared with Equation 2, this optimization can achieve up to 50% performance improvement in theory.

D. Optimization for Generator

As shown in Fig. 5(b), in our task parallelism version, t_n generation procedure has to wait until t_v generation procedure finishes, which decreases the overall throughput. *Synchronizer* also waits for the output of *Edge Validator*, although all visited bits of p_o are ready. Therefor, we carry out optimizations on *Generator* module.

Generator module is split into t_v *Generator* and t_n *Generator*. Once a new p_o is generated, it will be copied so that the source and the copy of p_o can be streamed into different FIFOs of two generators separately. Both t_v Generator and t_n Generator can start to work while *Synchronizer* starts to collect partial results at the same time. This optimization is achieved by copying data and using more on-chip resources (e.g., FIFOs). Thanks to the loop parallelism characteristic of FPGA, the cost of copy of p_o does not decrease the performance. And we analyze the total cycles of this optimized version L_{sep}. All modules execute concurrently. As a result, the minimum cycles we can achieve is as follows:

$$L_{sep} \approx N + \max(N, M) \tag{4}$$

Compared with equation 3, this optimization can achieve at most 33% performance improvement theoretically.

VII. Experiments

We present the results of our performance studies in this section. We first introduce the experimental setup of the experiments. Then, we investigate the necessity of CST partition and evaluate the effectiveness of our software and hardware optimizations, followed by the comparison with the state-of-the-art algorithms. We also evaluate our algorithm on a billion-scale graph to test the scalability.

Algorithms. We compare two state-of-the-art GPU-based solutions: GSI [39] and GpSM [35]. We do not comapre PBE [16] because it is unable to handle labeled graph. According to the latest survey [33], we compare other three state-of-the-art CPU-based algorithms: CFL [13], DAF [17], CECI [12] and five versions of our algorithm:

- *FAST-DRAM*: the algorithm fetches data from DRAM without any other optimizations.
- *FAST-BASIC*: the algorithm fetches data from on-chip memory without any other optimizations (Section VI-A).
- *FAST-TASK*: *FAST-BASIC* algorithm boosted by the task parallelism optimization (Section VI-C).
- *FAST-SEP*: *FAST-BASIC* algorithm boosted by the both task parallelism and task generator separation optimization (Section VI-D).
- *FAST-SHARE*: *FAST-SEP* algorithm where the host side, i.e. CPU, shares some matching tasks (Section V-C).

Among the five versions, we choose *FAST-SHARE* as the final version of our algorithm, denoted as FAST. The parallel version of DAF and CECI are also evaluated, denoted as DAF-8 and CECI-8 respectively, which run on 8 CPU threads. For all other algorithms, we use only one CPU thread.

Setup. We implement FAST in C++ on an Alveo U200 Data Center Accelerator Card, equipped with 64GB off-chip DRAM, 35MB on-chip BRAM, and communicates with the host through PCIe gen3 $\times$ 16. It runs at 300 MHz on the FPGA card. All experiments are conducted on a machine equipped with an 8-core Intel Xeon E5-2620 v4 CPU (2.1GHz), 250G host memory, NVIDIA Tesla V100 (5120 streaming processors, 16GB global memory), running Ubuntu 16.04.

TABLE III
CHARACTERISTICS OF DATASETS.

Name	$\|V_G\|$	$\|E_G\|$	d_G	D_G	# Labels
DG01	3.18M	17.24M	10.84	464,368	11
DG03	9.28M	52.65M	11.34	1,346,287	11
DG10	29.99M	176.48M	11.77	4,282,812	11
DG60	187.11M	1.25B	13.33	26,639,563	11

Datasets. The datasets commonly used in previous works [13], [17], [18], [29] are composed of small-scale data graphs (e.g., Yeast with 3.11K vertices and 12.51K edges) and large queries (e.g., 200 vertices), whereas the data graphs are usually very large and the queries are relatively small in real-world workloads nowadays. Therefore, we adopt the LDBC social network benchmarking (*LDBC-SNB*) [7] in our experiment to simulate real-world workloads. The LDBC-SNB benchmark serves as an industry-standard benchmarking and provides a data generator that generates a synthetic social network together with a set of benchmarking tasks, in which many tasks are subgraph matchings.

We list the datasets and their statistics in Table III. These datasets are generated simulating a real social network akin to Facebook with a duration of 3 years. The dataset's name, denoted as DGx, represents a scale factor of x.

Queries. We use the queries in [24], as shown in Fig. 6. The queries are selected from the LDBC-SNB's complex tasks with some adaptions, including only keeping the node types as labels and removing multi-hop edges in order to conform with the subgraph matching problem studied in this paper.

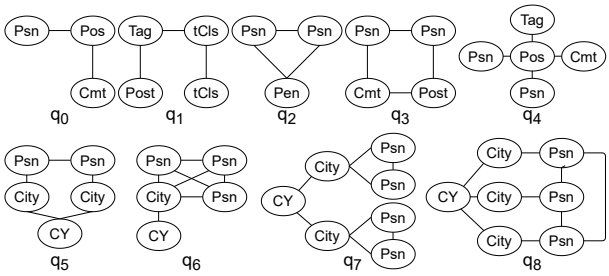


Fig. 6. The Queries

Metrics. To evaluate an algorithm, we measure the execution time in milliseconds. We set a time limit of 3 hours for each query. Each query is run three times and the *average* time is reported. We denote the execution time of queries with timeout as 'INF' and queries running out of memory as 'OOM'.

A. The Necessity of CST Partition

We partition CST in order to store it in BRAM instead of DRAM on FPGA because of the much higher read latency of DRAM. On the other hand, the random read of CST leads to the impossibility to prefetch the data from DRAM into BRAM. We compare the elapsed time of *FAST-DRAM* and *FAST-BASIC* to verify the necessity of CST partition.

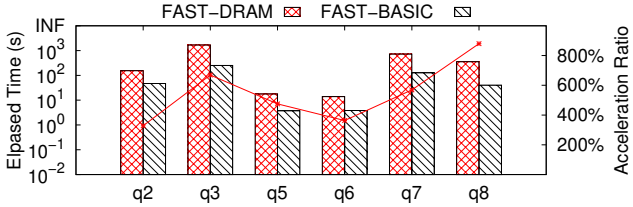


Fig. 7. Elapsed time of *FAST-DRAM* and *FAST-BASIC* for DG10

As shown in Fig. 7, the results indicate that *FAST-BASIC* outperforms *FAST-DRAM* for all the queries in both DG03 and DG10. Despite the initial overhead to fetch data from DRAM to BRAM, *FAST-BASIC* achieves about 5.0x speedup compared with *FAST-DRAM* on average. The speedup is close to the ratio of the read latency. Moreover, it is confirmed by the growing speedup (4.50x for DG01, 5.18x for DG03, and 5.93x for DG10) that the initial transmission overhead has a decreasing impact on the overall performance for a larger graph. These results show the necessity to partition CST structure to avoid direct data access to DRAM.

We use S_{CST} and S_G to denote the size of all CST partitions and data graph, respectively. Fig. 8 illustrates the number of CST partitions and $\frac{S_{\text{CST}}}{S_G}$, As expected, the number of CST partitions increases for larger data graphs. And $\frac{S_{\text{CST}}}{S_G}$ keeps stable for most queries while the data graph grows ($\frac{S_{\text{CST}}}{S_G} < 60\%$ for all queries). The rapid growth of $\frac{S_{\text{CST}}}{S_G}$ in q_7 from DG03 to DG10 is due to the rapid increase in the number of embeddings. Theses results confirm the scalability of our CST partition mechanism. Moreover, we conducted experiments about the impact of k in our technical report [21]

B. Evaluating Optimization Techniques

In this section, we test the four versions of our algorithm: *FAST-BASIC*, *FAST-TASK*, *FAST-SEP* and *FAST-SHARE* to evaluate our software and hardware optimizations.

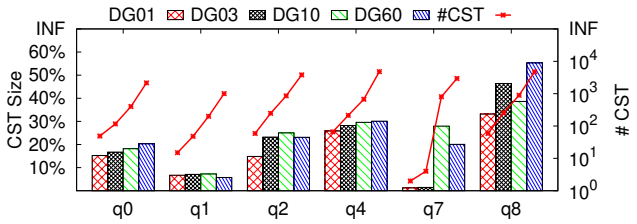


Fig. 8. The number and total size of partitioned CST

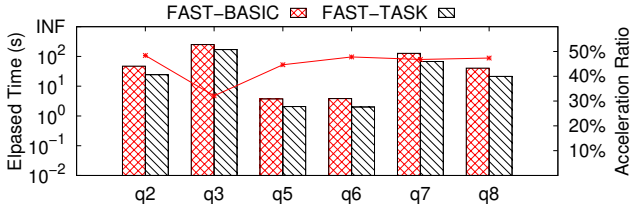


Fig. 9. Elapsed time of *FAST-BASIC* and *FAST-TASK* for DG10

Effectiveness of *Task Parallelism*. *FAST-BASIC* only adopts the loop pipeline mechanism of FPGA, while *FAST-TASK* introduces task parallelism so that full execution modules are allowed to operate in parallel. From the acceleration ratio in Fig. 9, we can see that the task parallel optimization achieves up to 50% improvement (e.g. q_8). The theoretical improvement of task parallelism is discussed in Section VI-C. From Equation (2) and Equation (3), we can see that the task parallelism optimization achieves better performance for dense queries whose M is larger than N. The acceleration ratio of q_3 is much lower than other queries because of its much higher $\frac{N}{M}$ (about 2 for $q3$ and close to or lower than 1 for other queries).

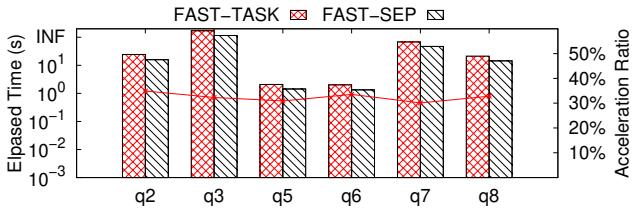


Fig. 10. Elapsed time of *FAST-TASK* and *FAST-SEP* for DG10

Effectiveness of *Task Generator Separation*. The task parallelism allows all modules to execute concurrently. However, it is limited by the first module *Generator* to generate two kinds of tasks in parallel so that the following modules can start to work at the same time. *FAST-SEP* solves this problem by using more on-chip resources and duplicating data. Compared with the average elapsed time of *FAST-TASK*, *FAST-SEP* achieves about $30\% - 40\%$ improvements (e.g. $q8$). The effectiveness of *Task Generator Separation* is consistent with our cycle analysis in Equation 3 and Equation 4. Moreover, when $\frac{N}{M} > 1$, *Task Generator Separation* achieves the best improvements.

Effectiveness of *Software Scheduler*. After partitioning CST, CPU becomes idle, which can be utilized to share some matching tasks. We propose a workload estimation method of CST and restrict the proportion of the total workload of matching tasks assigned to CPU from exceeding a threshold δ. We evaluate the effectiveness of *software scheduler* by varying δ. The results in Fig. 11 indicate that this optimization achieves biggest improvements when $delta = 0.1$ (e.g. 20% for DG01).

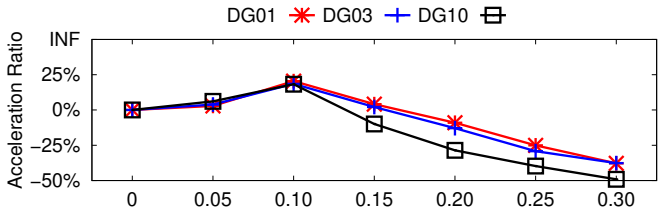


Fig. 11. Average acceleration ratio varying δ

The reason that this optimization achieves more than 10% improvements ($\delta = 0.1$ which means the host side shares about 10% matching tasks) is as follows: Considering CST that can not be fully offloaded into BRAM, in *FAST-SEP*, we have to partition it until it meets the size constraints; in *FAST-SHARE*, we may directly assign it to CPU, reducing the cost of partitioning. Moreover, it can be seen from the figure that the CPU becomes the bottleneck when $delta > 0.15$.

C. Comparing with Existing Algorithms

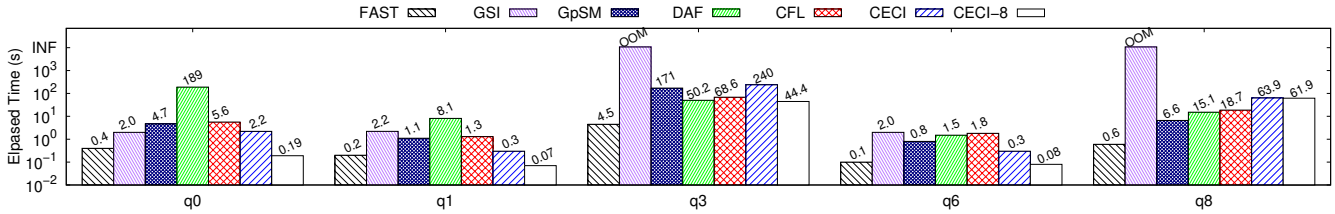
We then evaluate FAST against the existing algorithms, GSI [39], GpSM [35], CFL [13], DAF [17] and CECI [12]. All the source code comes from the original authors and is also implemented in C++. Fig 12 shows the experimental results. Each query show similar trend in different data graphs, so we only demonstrate the results of five queries for each data graph due to the space limit. Full results can be found in our technical report [21] FAST outperforms all other algorithms for all the queries and achieves 24.6x average speedup. Specifically, FAST outperforms GSI by up to 36.6x (q_6 in DG01), outperforms GpSM by up to 38.0x (q_3 in DG01), outperforms CFL by up to 191.0x (q_8 in DG10), outperforms DAF by up to 462.0x (q_0 in DG01) and outperforms CECI by up to 150.0x (q_8 in DG10).

We noticed that the GPU-based solutions do not show better performance over CPU-based algorithms for some queries. More critically, both GSI and GpSM are only able to handle the graphs that can be fit into the GPU memory. So they both fail to solve all the queries. The reason why GSI has a higher memory cost is that GSI pre-allocates enough memory space instead of joining twice like GpSM to avoid the conflicts when each processor writes results to memory in parallel.

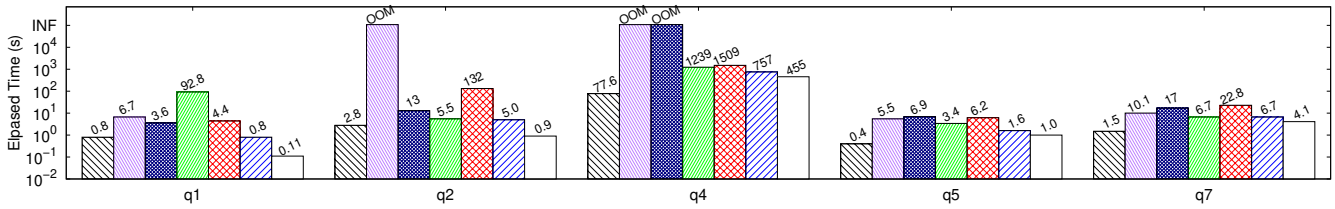
Both DAF and CECI adopt the intersection-based method which makes them performs better than edge verification method CFL-Match in most cases. Although FAST adopts the edge verification method, it can finish edge verification in one cycle thanks to our pipelining design on FPGA, which makes its cost even less than the intersection-based method on CPU.

Another trend in Fig. 12 is that as data size grows, the acceleration ratio of FAST compared with other three CPU-based algorithms also increases, e.g., for $q3$, the average rate is 26.0x, 33.0x and 59.0x and for q_8, the average acceleration rate is 59.0x, 86.0x and 121.0x in DG01, DG03 and DG10, respectively. It is because the cost of edge verification in FAST remains one cycle while the cost in each *recursive call* grows in other three CPU-based algorithms as the data size grows.

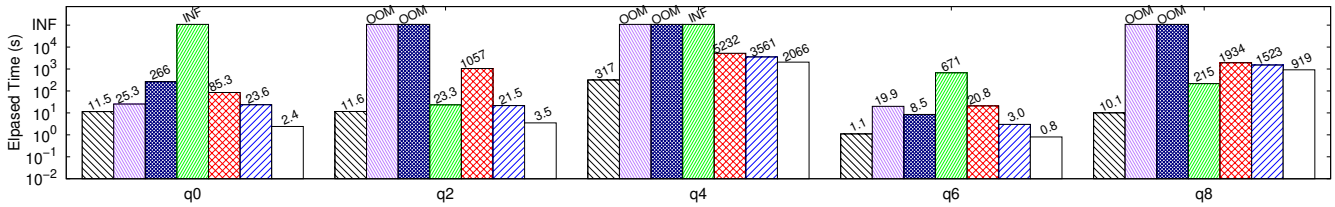
For full comparison, the parallel version DAF-8 and CECI-8 are also evaluated. However, DAF-8 encounters out of memory error when processing DG03 and DG10. So we only prsent the results of CECI-8 in Fig. 12. The average acceleration rate



(a) DG01



(b) DG03



(c) DG10

Fig. 12. Elapsed time of CFL-Match, CECI, DAF and FAST

of FAST compared with CECI-8 is 5.79x, 8.51x and 9.31x in DG01, DG03 and DG10, respectively.

We adopt the path-ordering matching order similar to CFL. To evaluate the impact of matching orders, we test FAST with the following orders: (1) CFL's order; (2) DAF's order; (3) CECI's order; (4) all other random connected orders. The results (averaged over all queries) are illustrated in Fig. 13. For each query, we extract the minimum, average and maximum elapsed time denoted as BEST, AVG and WORST orders, respectively. It can be seen from the figure that the average elapsed time of FAST with CFL's, DAF's and CECI's orders is very close to each other. The FAST with WORST matching order can still outperform CFL, CECI and DAF (by 9.6x, 11.1x and 36.3x, respectively).

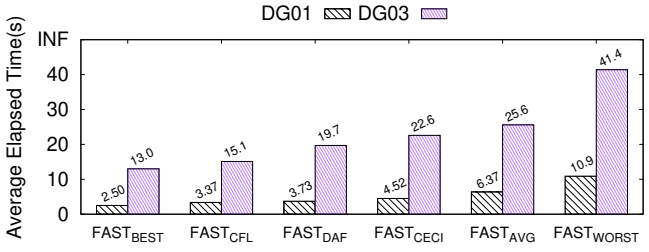


Fig. 13. The elapsed time of FAST with different matching orders

D. Scalability Testing

In this subsection, we evaluate the scalability of our FAST algorithm by using a billion-scale graph DG60.

Varying scale factor. We run all algorithms on the DG01, DG03, DG10 and DG60. All the other three algorithms fail

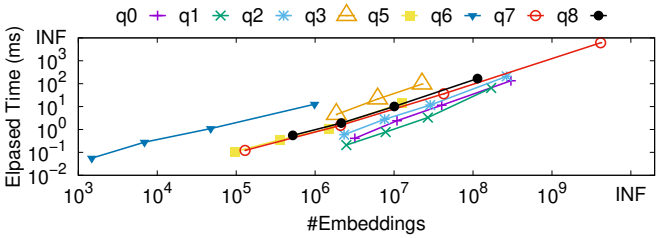


Fig. 14. Scalability Testing of FAST (vary x)

to finish a single query for the DG60. *CECI* has a segment fault during execution. *CFL-Match* uses an adjacency matrix representation of the data graph to overcome the overhead of edge verification, resulted in out of memory errors for large graphs like DG60. As for *DAF*, it encounters overflow errors during execution. The problem is caused by the much fewer labels of the *LDBC* datasets (i.e. 11 labels) which makes the search space larger. FAST completes all queries successfully. The experimental result of FAST is illustrated in Fig. 14. The elapsed time increases linearly with respect to the number of embeddings as the scale factor of x for DGx grows.

Varying $|E(G)|$. We keep all vertices and sample 20%, 40%, 60%, and 80% edges of DG60 uniformly to further test the scalability of FAST. Fig. 15 indicates that the **average** elapsed time per embedding has no apparent changing as $|E(G)|$ increases, which verifies the scalability of FAST. The reasons for high elapsed time per embedding for $q5$, $q6$, and $q8$ in the 20% sample are as follows: (1) The number of embeddings is very small for these queries, e.g., 12 for $q6$ and 36 for $q8$. (2) The cost of data transfer and index construction affects overall performance more apparently, when $E(G)$ is small.

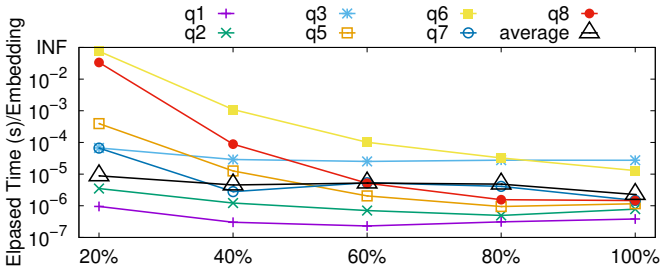


Fig. 15. Scalability Testing of FAST (vary $|E(G)|$)

E. Discussion

FAST algorithm can be easily extended to multi-FPGA environments. Each CST structure is an independent and complete search space. Combined with our workload estimation method, the CPU can assign the CST structure to the FPGA with the minimum total workload and collect final results after all the FPGAs complete their tasks. One interesting future work is to combine FAST in the distributed environment to accelerate distributed subgraph matching.

VIII. Conclusion

In this paper, we present the first CPU-FPGA co-designed framework to accelerate subgraph matching. Our BRAM-only matching process significantly reduces the costly data transfer between BRAM and DRAM on FPGAs. Moreover, with the workload estimation method of CST, our framework can be potentially extended to multi-FPGA environment. The experimental results demonstrate that our framework significantly outperforms the state-of-the-art algorithms. In the future, we will investigate integrating FAST into graph database systems and RDF engines to accelerate subgraph queries.

References

[1] Project catapult. [Online]. Available: https://www.microsoft.com/en-us/research/project/project-catapult/.

[2] Amazon ec2 f1 instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[3] Intel fpgas power acceleration-as-a-service for alibaba cloud. [Online]. Available: https://newsroom.intel.com/news/intel-fpgas-power-acceleration-as-a-service-alibaba-cloud/.

[4] Tencent fpga cloud server. [Online]. Available: https://cloud.tencent.com/product/fpga.

[5] Fpga-accelerated cloud server. [Online]. Available: https://www.huaweicloud.com/en-us/product/fcs.html.

[6] Xilinx fpgas on the nimbix cloud. [Online]. Available: https://www.nimbix.net/xilinx/.

[7] Ldbc benchmark. [Online]. Available: http://ldbcouncil.org/benchmarks.

[8] Neo4j. [Online]. Available: https://neo4j.com/.

[9] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *PVLDB*, 11(6):691–704, 2018.

[10] M. Besta, M. Fischer, T. Ben-Nun, J. de Fine Licht, and T. Hoefler. Substream-centric maximum matchings on fpga. In *2019 ACM/SIGDA International Symposium on FPGA*, pages 152–161, 2019.

[11] M. Besta, D. Stanojevic, J. D. F. Licht, T. Ben-Nun, and T. Hoefler. Graph processing on fpgas: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[12] B. Bhattarai, H. Liu, and H. H. Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *2019 ACM SIGMOD*, pages 1447–1462, 2019.

[13] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *2016 ACM SIGMOD*, pages 1199–1214, 2016.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[15] N. Engelhardt and H. K.-H. So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *26th International Conference on FPL*, pages 1–4. IEEE, 2016.

[16] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *2020 ACM SIGMOD*, pages 1067–1082, 2020.

[17] M. Han, H. Kim, G. Gu, and etc. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *2019 ACM SIGMOD*, pages 1429–1446, 2019.

[18] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *2013 ACM SIGMOD*, pages 337–348, 2013.

[19] J. Hartmanis. Computers and intractability: a guide to the theory of np-completeness). *Siam Review*, 24(1):90, 1982.

[20] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *2008 ACM SIGMOD*, 2008.

[21] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng. FAST: FPGA-based Subgraph Matching on Massive Graphs. Technical report. Available: http://www.cse.unsw.edu.au/~zyang/FAST_Full.pdf.

[22] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[23] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *PVLDB*, 10(3):217–228, 2016.

[24] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, et al. Distributed subgraph matching on timely dataflow. *PVLDB*, 12(10):1099–1112, 2019.

[25] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2014.

[26] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. Subgemini: identifying subcircuits using a fast subgraph isomorphism algorithm. In *30th international Design Automation Conference*, pages 31–37, 1993.

[27] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, 2006.

[28] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *PVLDB*, 11(2):176–188, 2017.

[29] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[30] M. Serafini, G. De Francisci Morales, and G. Siganos. Qfrag: Distributed graph search via subgraph isomorphism. In *2017 Symposium on Cloud Computing*, pages 214–228, 2017.

[31] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[32] T. A. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological methodology*, 36(1):99–153, 2006.

[33] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *2020 ACM SIGMOD*, pages 1083–1098, 2020.

[34] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *25th Symposium on Operating Systems Principles*, pages 425–440, 2015.

[35] H.-N. Tran, J.-j. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*, pages 299–315. Springer, 2015.

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

[37] L. Wang, Y. Wang, and J. D. Owens. Fast parallel subgraph matching on the gpu. *HPDC*, 2016.

[38] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *2004 ACM SIGMOD international conference on Management of data*, pages 335–346, 2004.

[39] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. Gsi: Gpu-friendly subgraph isomorphism. In *2020 IEEE 36th ICDE*. IEEE, 2020.

[40] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2):340–351, 2010.

[41] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna. An fpga framework for edge-centric graph processing. In *15th ACM International Conference on Computing Frontiers*, pages 69–77, 2018.

[42] S. Zhou and V. K. Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th SBAC-PAD*. IEEE, 2017.

[43] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. Gstore: Answering sparql queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.