FASI: FPGA-friendly Subgraph Isomorphism on Massive Graphs

Xunbin Su
Peking University
Beijing, China
suxunbin@pku.edu.cn

Yinnian Lin
Peking University
Beijing, China
linyinnian@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

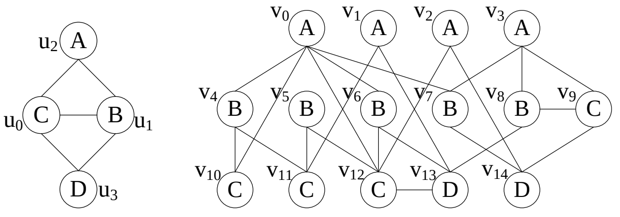
***Abstract*—Subgraph isomorphism plays a significant role in many applications, such as social networks and bioinformatics. However, due to the inherent NP-hardness, it becomes challenging to compute matches efficiently in large real-world graphs. Many researchers have attempted to solve this problem with the help of new hardware. Nevertheless, most of them focus on GPU. Due to the dataflow feature and burst I/O optimization, FPGA is a potential competitor to speed up subgraph isomorphism. However, there are very few subgraph matching algorithms on FPGA. In this paper, we present an efficient FPGA-friendly Subgraph Isomorphism algorithm `FASI`, designed on CPU-FPGA heterogeneous platform which leverages FPGA's features. Unlike the existing FPGA-based method `FAST`, we adopt the worst-case-optimal-join-based pipeline design. First, we propose an FPGA-friendly data structure `LPCSR` for efficient access to neighbor lists. Second, we offer a joint parallelized pipeline strategy to accelerate matching process. Third, we propose a memory coalescing mechanism and a space-saving pre-allocated write back strategy. Our experiments on both synthetic and real graphs show that `FASI` outperforms other state-of-the-art subgraph matching algorithms on CPU, GPU and FPGA.**

***Index Terms*—FPGA, Subgraph Isomorphism, WOJ, Pipeline**

I. Introduction

In recent years, subgraph matching has played an increasingly important role in many graph analysis tasks. It aims to find all distinct subgraphs of a data graph G that are isomorphic to a query graph Q. For example in Figure 1, given a query graph Q and a data graph G, there is one match of Q over G, i.e., $\{(u_0, v_{12}), (u_1, v_6), (u_2, v_0), (u_3, v_{13})\}$. In practice, subgraph matching (also known as subgraph isomorphism) has attracted much attention in academia and industry. It has been widely used in various domains, e.g., social network analysis [1]–[3], protein-protein interaction network analysis [4], [5], chemical compound search [6], graph pattern mining [7], [8] and RDF query processing [9], [10]. However, it is challenging to efficiently compute all matches of Q over a larger G since subgraph isomorphism is a classical NP-hard problem [11]. Therefore, speeding up subgraph matching on massive graphs is the focus of our work.

Extensive research has been conducted to find solutions for speeding up subgraph matching on CPUs [12]–[19] and most of them adopt the backtracking approach [20], [21]. The approach follows the idea of depth-first search, which recursively maps the next query vertex to a data vertex to get all matches. Although existing algorithms on CPUs propose many optimization techniques on matching orders, pruning rules, and index structure using various heuristic methods, the search space is still large when handling massive graphs. Meanwhile, general-purpose CPUs cannot provide a high degree of parallelism and flexible cache mechanism. As a consequence, asking for hardware assistance is a better choice.



(a) Query graph

(b) Data graph

Fig. 1. Subgraph Matching

Recently, many works have devoted much effort to speed up subgraph matching by leveraging massively parallel computation capability of GPUs [22]–[26]. They are proved to be effective in improving the performance of subgraph matching. In fact, FPGAs also have advantages over CPUs on parallelism, thanks to the data stream transfer without instruction decoding and pipeline processing. Meanwhile, compared to GPUs, FPGAs have larger on-chip memory and lower power consumption [27]. Therefore, FPGAs are often used for accelerating artificial neural networks for machine learning applications [28]–[31] and some common graph algorithms [32]–[34]. However, there are few subgraph matching algorithms developed on FPGA, except for `FAST` [35], the only existing FPGA-based subgraph matching solution.

`FAST` generates an auxiliary data structure `CST` (candidate search tree) as a complete search space when handling a query graph. Then it adopts an edge verification method to enumerate all matches. However, `FAST` suffers from the heavy overhead of generating `CST` on CPU at the query runtime. In addition, the edge verification method leads to lots of redundant work because the candidates of a query vertex are usually a lot and a large number of partial results do not need to be expanded.

Generally, there are three main challenges when developing subgraph matching algorithms on FPGAs:

- *Limited On-chip Memory.* The on-chip memory of up-to-date FPGAs includes Block RAM (BRAM) and Ultra-RAM (URAM). Because the size of total on-chip memory on FPGA is as small as only tens of megabytes, large graph data cannot be stored into on-chip memory directly. Thus, FPGA needs off-chip memory (DRAM) to store graph data, but how to optimize DRAM memory access to fetch a vertex's neighbors is critical in improving the performance. Furthermore, the size of intermediate results is quite large in the subgraph matching process. Traditional worst-case-optimal-join (WOJ) based subgraph matching systems [24], [36], [37] should materialize the whole intermediate results, which is infeasible to cache them on on-chip memory unless flushing them to DRAM, but writing to DRAM is quite slow.

- *Low Clock Frequency.* Because FPGA has a lower clock frequency than GPU and CPU (e.g., 300MHz vs. 2.4GHz), it needs higher parallelism to get better performance. This requires us to design high parallelism subgraph matching algorithms using FPGA's features, such as dataflow. Existing backtracking-based solutions on CPU are difficult to be parallelized on FPGA.
- *Parallel Write Conflicts.* When multiple pipelines need to write their corresponding results to DRAM, the same address may be written, leading to writing conflicts. Therefore, how to ensure the correctness of the concurrently outputting results is also a question to consider.

To address the above challenges, we propose an efficient FPGA-friendly Subgraph Isomorphism algorithm `FASI`, which conducts the whole subgraph matching process on FPGA using the pipeline evaluation strategy. Different from `FAST`, we adopt the WOJ strategy on FPGA. To speed up the process, we first propose an FPGA-friendly data structure (`LPCSR` in Section V) to support efficient neighbor list fetching. We call an algorithm or a data structure FPGA-friendly since they take advantage of some specific optimization techniques for FPGA (such as burst I/O and dataflow features in FPGA) to get better performance. Moreover, we propose a pipeline evaluation to parallelize the whole subgraph matching process using FPGA's dataflow, which increases the parallelism and reduces on-chip memory requirement. More detailed discussions can be found in Section IV-B. We summarize our contributions as follows:

- We propose an efficient CPU-FPGA co-designed subgraph matching algorithm using WOJ-based pipeline join. We exploit FPGA's dataflow feature to implement a left-deep-tree-based pipeline join accelerator for WOJ. To reduce random memory access and increase continuous memory access to DRAM, we design a coalescing mechanism on FPGA, which exploits burst I/O on FPGA using AXI protocol.
- We propose an FPGA-friendly data structure `LPCSR` to represent labeled graphs. It not only reduces memory access to DRAM when getting a vertex's neighbors but also improves the spatial locality of memory access, which further improves the performance of memory coalescing.
- To address writing conflicts on FPGA, we propose a space-saving pre-allocated write back strategy, which increases a little space overhead and eliminates writing conflicts.
- We conduct experiments on both synthetic and real graph datasets. The results show that our algorithm outperforms the state-of-the-art algorithms by several orders of magnitude (e.g., up to 11.35x over the CPU-based solution `CECI` [38], up to 33.95x over the only existing FPGA-based solution `FAST` and up to 53.9x over `GpSM` [23]).

II. Preliminary

A. Problem Definition

In this paper, we focus on undirected vertex-labeled graphs, although it is trivial to extend our approach to handle directed and edge-labeled graphs. We define the subgraph isomorphism search as follows.

Definition 1 (Graph)**.** *A graph G is a tuple $G = \{V, E, L\}$, where $V(G)$ is a set of vertices, $E(G) \subset V(G) \times V(G)$ is an edge set and L is a labeling function that assigns vertex labels.*

Definition 2 (Subgraph Isomorphism)**.** *Given a query graph Q and a data graph G, Q is subgraph isomorphic to G if and only if an injective mapping function M from $V(Q)$ to $V(G)$ exists such that $\forall u \in V(Q), L(u) = L(M(u))$ while $\forall (u, u') \in E(Q), (M(u), M(u')) \in E(G)$, where $M(u)$ is the mapped data vertex of u.*

There may be various subgraph isomorphisms in G, each of which is referred to as an *embedding* of Q in G.

Definition 3 (Subgraph Isomorphism Search)**.** *Given a query graph Q and a data graph G, the subgraph isomorphism search problem is to find all embeddings of Q in G.*

Due to the NP-hardness of subgraph isomorphism, we resort to hardware assistance. In this paper, we propose an efficient FPGA-based solution for subgraph isomorphism search. Table I lists the frequently-used notations throughout the paper.

TABLE I
Frequently used notations.

Notation	Description
Q and G	Query graph and data graph
$V(G)$ and $E(G)$	Vertex set and edge set of G
$L(G)$	A labeling function of G
$d(u)$	Degree of vertex u
$M(u)$	Mapping of u in an embedding M
$C(u)$	Candidate list for query vertex u
$N(v, l)$	Neighbor list of vertex v with label l

B. Worst Case Optimal Join

Generally, subgraph matching can be solved by a series of join operations. A join operation refers to extending the partial embeddings by matching a new query edge or vertex, while the former is called *binary join* and the latter is *worst-case optimal join (WOJ).*

This paper focuses on developing an FPGA-based WOJ due to two reasons: First, the WOJ provides a better worst-case performance guarantee; Second, the binary join cannot avoid parallel writing conflicts except for *join-twice output* *scheme* [39] in the parallel processing. WOJ can alleviate that by forecasting the cardinality of extending each partial embedding [24], and we propose a space-saving pre-allocated strategy (in Section VI). A typical WOJ algorithm enumerates over vertices of query graph Q in a given matching order $O(Q) = \{o_0, ...o_k, ...o_{|V(Q)|-1}\}$. Let P_k be the set of partial embeddings consisting of $\{o_0, ..., o_k\}$. At each iteration, a join operation extends all partial embeddings in P_k by calculating the intersection among the candidate set of o_{k+1} and each of the neighbor sets of o_i for every $(o_i, o_{k+1}) \in E(Q)$, where $i \leq k$. As shown in Figure 2, we match the query graph in Figure 1(a) and the current partial embedding set is $P_2 = \{\{v_9, v_8, v_3\}, \{v_{10}, v_4, v_0\}, \{v_{12}, v_6, v_0\}\}$ and the next vertex is $o_3 = u_3$ with $C(u_3) = \{v_{13}, v_{14}\}$. In this case, extending the third embedding can be formalized as $\{v_{12}, v_6, v_0\} \times \{N(v_{12}) \cap N(v_6) \cap C(u_3)\} = \{\{v_{12}, v_6, v_0, v_{13}\}\}$.

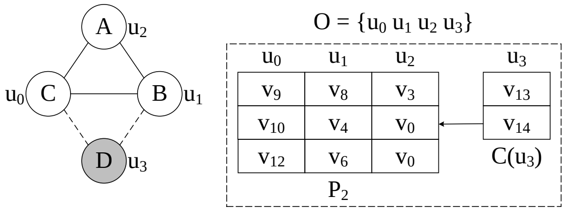


Fig. 2. Worst-case optimal join on graphs

C. FPGA Architecture

A field-programmable gate array (FPGA) is an integrated circuit made from configurable logic and memory blocks, which can implement any user-defined logic by connecting logic and memory units with configurable wires. It usually serves as a co-processor of the CPU for accelerating computation. As shown in Figure 3(a), CPU and FPGA are connected via PCIe and equipped with private memory units (main memory for CPU and global memory for FPGA). In this paper, we focus on two optimization techniques for efficient FPGA-based algorithms: *burst I/O* and *dataflow*.

Burst I/O. Processing a memory request on FPGA follows the AXI protocol. Each memory request has an AXI head for auxiliary information. Figure 3(b) illustrates an example of three memory read requests. Without burst I/O, they are processed one by one. However, the AXI protocol allows us to aggregate consecutive memory requests to DRAM in burst mode, handling continuing memory reads after parsing the first AXI head until receiving all required data. This feature invokes us to design a memory coalescing technique to aggregate memory requests as much as possible by tolerating redundant memory access.

Dataflow. The dataflow in FPGA provides a possibility of implementing the pipeline execution hardware-level model-based query processing. A database query is decomposed into a pipeline of operations, such as the left-deep join tree [40], in which intermediate results are passed to the next operation as soon as they are generated. The traditional query pipeline evaluation is at the software level, which is first transferred to a set of CPU instructions. However, with the dataflow model in FPGA, we can directly create on-chip computation modules, design the dataflow between them and skip the overhead of generating, fetching, and decoding instructions as shown in Figure 3(c). This feature motivates us to design an end-to-end FPGA kernel containing multiple concatenated join operation modules during subgraph matching. Partial embeddings are passed in a streaming way between these modules in the dataflow model. To the best of our knowledge, we are the first to propose an FPGA dataflow-based query pipeline implantation in graph databases, especially for WOJ.

III. Related Work

A. CPU-based Subgraph Matching

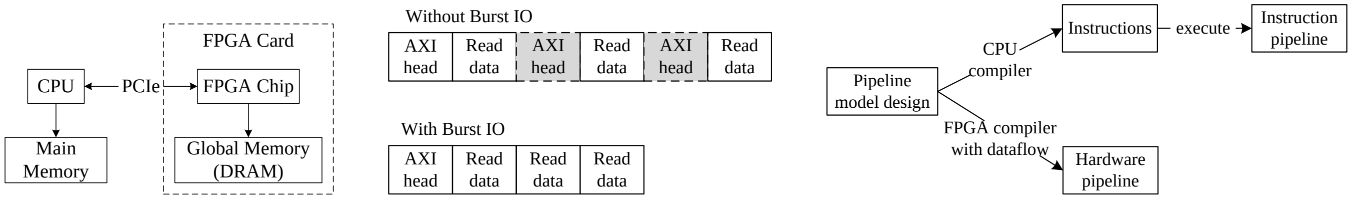
The early study of subgraph matching can be traced back to Ullmann's backtracking algorithm [20], which uses a depth-first search strategy to match query vertices. Many subsequent works [12], [14], [15], [17] focus on reducing the search space by different optimization strategies. A comprehensive survey on these algorithms has been conducted by Lee et al. [21]. Later, `TurboISO` [18] and `BoostISO` [16] exploit vertex similarity to merge query vertices and data vertices to reduce redundant computations, respectively. `CFL-Match` [19] develops a Core-Forest-Leaf decomposition and proposes CPI structure for pruning. `CBWJ` [41] optimizes subgraph matching by combining binary and worst-case optimal joins. `CECI` [38], and `DAF` [42] replace the edge verification method with the set intersection strategy to find candidates faster. `RapidMatch` [43] combines exploration-based and join-based methods. However, these CPU-based solutions suffer from performance issues for large graphs due to low parallelism.

B. GPU-based Subgraph Matching

The earlier works [25], [44] on GPU try to transplant existing CPU-based subgraph matching algorithms. Unfortunately, these backtracking-based algorithms suffer from warp divergence and uncoalesced memory access on GPU, as Jenkins et al. [45] analyzed. Later, `GpSM` [23], and `GunrockSM` [22] adopt the breadth-first search strategy for higher parallelism on GPU, which demonstrates better performance. They both adopt the edge-oriented join strategy for matching and the two-step output scheme to avoid writing conflicts, which have increased the computation workload. `GSI` [24] proposes a Prealloc-Combine approach, which uses the vertex-oriented join strategy and pre-allocates enough memory space to avoid joining twice. However, these GPU-based solutions cannot effectively handle large graphs that are not fit into GPU's global memory.

C. FPGA-based Subgraph Matching

`FAST` [35] is the first and only existing FPGA-based subgraph matching algorithm. It proposes an auxiliary data structure `CST` to serve as a complete search space and a partitioning strategy to make each `CST` fit into FPGA's BRAM. It adopts the edge verification method to enumerate matches on FPGA, which brings many redundant verifications. Moreover, `CST` needs to be rebuilt on the CPU for each query graph. This brings extra runtime overhead and affects the performance.



(a) Basic architecture (b) Burst IO (c) dataflow

Fig. 3. FPGA Architecture

IV. Overview of FASI

Our FASI system consists of the CPU host and the FPGA kernel, as shown in Figure 4(a). The CPU host preprocesses a data graph G and generates candidates and a join order O for a given query graph Q. The FPGA kernel, which has multiple processing engines (PEs) and is PCIe-attached to the CPU host, receives input from the CPU host and performs specific subgraph matching tasks to find all matches of Q in G.

A. The CPU Host

At the offline processing, for better memory coalescing in the FPGA kernel at the running time, the CPU host reorders G to make vertices whose neighbor lists are often visited together as adjacent as possible and the amount of wasted read data (during burst read) as small as possible. More details will be discussed in Section VI-A2. After reordering, we construct an FPGA-friendly data structure LPCSR (proposed in Section V) for G and offload LPCSR to FPGA's off-chip memory through PCIe bus. The entire data preprocessing phase is offline.

At the query runtime, given a query graph Q, we use two simple yet effective filtering strategies (LDF and NLF[1]) to generate candidate lists $C(u)$ for each query vertex u. Note that many sophisticated candidate generation strategies [19], [38], [42] can also be used in this stage, and we focus on the FPGA-based subgraph matching process, and the candidate generation strategy is orthogonal to our method. We use a heuristic strategy to determine the join order O of Q, which is often used in other subgraph matching algorithms [24], [38] as well. Specifically, we greedily select $u_{next} = argmin_{u \in V(Q')} \frac{|C(u)|}{d(u)}$ as the next query vertex, where $C(u)$ is generated by LDF and NLF, $V(Q')$ is the set of query vertices that have not yet been matched. Generally, at the running time, the CPU host transfers the candidate lists and the join order to FPGA's off-chip memory through the PCIe bus and launches the FPGA kernel to compute subgraph matching.

B. The FPGA Kernel

One of our major technique contributions is to fully utilize FPGA's dataflow feature to implement the *pipelined evaluation*. We follow the generic join developed by Ngo et al. [47], which evaluates queries using a *vertex-at-a-time* strategy. If we adopt the *materialized evaluation* in the traditional WOJ systems (such as GraphFlow [36], EmptyHeaded [37], etc), we have to flush intermediate results to DRAM in each join step since the on-chip memory size is too small to cache them even though we use both BRAM and URAM. This is time-consuming because of the limited bandwidth of DRAM. FAST [35] partitions CST to make sure each CST partition does not exceed the on-chip storage capability, but it launches multiple cross-device data transfers and does not make full use of PCIe bandwidth. Our *pipelined evaluation* on FPGA not only reduces the requested buffer size significantly but also hides the transmission cost between CPU and FPGA. Besides the pipeline parallel processing, we build multiple PEs, which process distinct parts of the start vertex's candidates in parallel.

We describe the PE structure in Figure 4(b). It consists of a candidate reader ①, a series of extension modules ② and a result writer ③. To reduce the time-consuming data transfer between DRAM and BRAM, we maintain a series of FIFO buffers using both BRAM and URAM in our design. URAM has more storage capacity and BRAM has faster access speed. Therefore, we use BRAM to store the neighbor lists and candidate vertices accessed from DRAM and use URAM to cache the partial intermediate results avoiding frequently flushing them to DRAM. The candidate reader reads a batch of candidate vertices into the candidate buffer on BRAM. Along the given join order, each extension module expands the intermediate results of the previous step using the next query vertex and writing the newly generated intermediate results into the intermediate result buffer on URAM. There is an intermediate result buffer between each two extension modules. The result writer will write the final results in the final result buffer to DRAM.

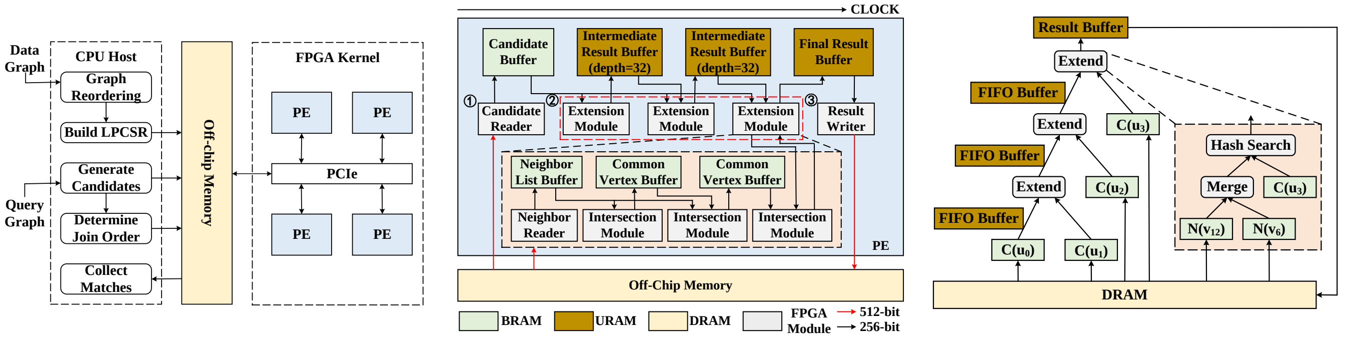
(1) Specifically, based on the given join order O, each PE can be represented as a *left-deep join tree* for WOJ as follows:

Definition 4 (Left-Deep Join Tree for WOJ)**.** *Given a join order $O = \{u_0, ..., u_{n-1}\}$ and the candidate lists $C(u_i)$ for each query vertex u_i ($i = 0, ..., n-1$), the left-deep join tree T for WOJ is defined as follows:*

- *T is a left-deep binary tree with n leaf nodes f_i, each of which corresponds to a query vertex u_i;*
- *Each inner node o_i ($i = 1, ..., n-1$) corresponds to a subquery (of the query graph Q) induced by the first $i+1$ query vertices in Q, denoted as $\mathcal{Q}_{[0,i]}$.*

Figure 4(c) shows an example of the left-deep join tree for the query graph Q in Figure 1(a) and Algorithm 1 gives an FPGA-based pipeline evaluation for subgraph matching (basic version). Assume that the join order is $\{u_0, u_1, u_2, u_3\}$. Each

[1] The label and degree filtering (LDF) and the neighbor label frequency filtering (NLF) are defined in [46].



(a) System overview

(b) Structure of processing engine (PE)

(c) Left-deep join tree of query Q in Figure 1(a)

Fig. 4. FASI Architecture

leaf node corresponds to one query vertex u_i and its candidate list $C(u_i)$. Each inner node o_i corresponds to an extension module (Lines 13-14 in Algorithm 1) between matches of subquery $\mathcal{Q}_{[0,...,i-1]}$ (i.e., $M(\mathcal{Q}_{[0,...,i-1]})$) with $C(u_i)$. We adopt eager pipelining, in which the lower level operator eagerly passes the results to the higher level one (Lines 20-22) and does not wait for the higher level one to request the results. Thus, we maintain a buffer P_i (in URAM) (Lines 2-4) to cache the results generated by the lower-level operator.

Besides the pipeline-based parallelization can improve the performance, pipeline-based evaluation can address the scalability issue due to limited on-chip storage capability. Although the number of partial embeddings is large and the buffer size is limited by the total on-chip storage capability (including BRAM and URAM, <35M in our case), due to the pipeline design, the on-chip memory requirement is reduced greatly. Thus, we can implement the whole subgraph matching process on FPGA, avoiding swapping intermediate results to DRAM and reduce the data dependency to maximize the pipelining parallelism and hide the data transmission cost.

(2) As mentioned above, each inner node of the *left-deep join tree* T corresponds to an extension module, which includes a neighbor reader and a series of intersection modules. The neighbor reader accesses neighbor lists from DRAM, and the intersection modules conduct set intersection operations. Specifically, for each extension module, we also design it as a *set-intersection tree* as follows:

Definition 5 (Set-intersection Tree)**.** *Given a partial match m corresponding to the subquery $\mathcal{Q}_{[0,...,i-1]}$ and the next query vertex u_i with label l, assume that u_i is connected to p query vertices in $\mathcal{Q}$, denoted as $[u_{j_0}, ..., u_{j_{p-1}}]$ ($p < i$). The set-intersection tree corresponding to extending m with u_i is a left-deep binary tree, where each leaf node corresponds to a query vertex u_j and its label-constrained neighbors $N(u_j, l)$ except for the right-most leaf node that corresponds to the next query vertex u_i and its candidates $C(u_i)$.*

The dashed box of Figure 4(c) is an example of the set-intersection tree corresponding to the last extension module when evaluating the query graph Q in Figure 1(a). Specifically, we propose a pipeline set intersection algorithm on FPGA in Algorithm 2. In each step, we do pairwise intersection (Lines 5-15 in Algorithm 2). We also adopt the eager pipelining (Lines 8, 12, 15) and maintain a common vertex buffer B_i (Lines 1-3). In the first step, we conduct *merge-based* intersection. Once finding one common vertex, we push it into the upper buffer (Lines 6-8). The upper operator fetches each vertex v from the buffer and checks the existence over another vertex's neighbor list (i.e., the right child leaf node) by *binary search* (Lines 9-12). The top-level module checks the existence of the candidates of the next query vertex using *hash search* since we represent its candidates by the bitmap (Lines 13-15). Once we get a common vertex v in the top-level intersection in Algorithm 2, we concatenate the original size-i partial match m with v (i.e., $m \oplus v$) and push it into P_i (in Algorithm 1).

Algorithm 1: FPGA-based pipeline WOJ (basic)

```
   Input: Query Graph $Q$, Data Graph $G$ and a join order
          $\{u_0, ..., u_{n-1}\}$
   Output: All matches of $Q$ over $G$, denoted as $M$.
1  $M \leftarrow \varnothing$ ; /* $M$ is located at off-chip DRAM */;
2  /* $P_i$ is located at on-chip URAM */;
3  for $i \leftarrow 0$ to $n-1$ do
4      Let $P_i$ be the partial result buffer, set $P_i \leftarrow \varnothing$ ;
5  for $i \leftarrow 0$ to $n-1$ pipeline do
6      $u_i, C(u_i) \leftarrow$ next query vertex and its candidates;
7      if $i$ == 0 then
8          foreach $v \in C(u_i)$ do
9              $P_0$.enqueue($v$);
10     else
11         foreach partial match $m \in P_{i-1}$ pipeline do
12             $P_{i-1}$.dequeue($m$);
13             /*Call Algorithm 2 to extend $m$ with $u_i$*/;
14             Let $B = m \times Extend(m, u_i)$;
15             /*the last query vertex*/;
16             if $i$ == $n-1$ then
17                 /*Add matches $B$ into $M$ in DRAM */;
18                 $M = M \cup B$ ;
19             else
20                 /*Push generated partial matches into $P_i$*/;
21                 foreach $m' \in B$ do
22                     $P_i$.enqueue($m'$);
23 return $M$;
```

V. Data Structure: LPCSR

Due to I/O irregularity of graph data, memory access cost bottlenecks the performance of subgraph matching on FPGA. In this section, we propose a FPGA-friendly data structure

```
Algorithm 2: Extend($m$, $u_i$)
Input: Partial match $m[v_0, ..., v_{i-1}]$ corresponding to
       subquery $\mathcal{Q}_{[0,...,i-1]}$, next query vertex $u_i$ with label
       $l$, $u_i$ is connected to $p$ query vertices in $\mathcal{Q}$, denoted
       as $(u_{j_0}, ..., u_{j_{p-1}})$ $(p < i)$
Output: Mappings of $u_i$ satisfying edge constraints.
/* $B_i$ is located at on-chip BRAM */;
for $i \leftarrow 1$ to $p$ do
    Let $B_i$ be the temporal intersection, set $B_i \leftarrow \varnothing$ ;
Let data vertices $(v_{j_0}, ..., v_{j_{p-1}})$ (in $m$) match query vertices
  $(u_{j_0}, ..., u_{j_{p-1}})$ ;
for $i \leftarrow 1$ to $p$ pipeline do
    if $i == 1$ then
        Do merge intersection between $N(v_{j_0}, l)$ and
          $N(v_{j_1}, l)$ ;
        Push common vertices to $B_1$ once found ;
    if $1 < i < p$ then
        foreach vertex $v$ into buffer $B_{i-1}$ do
            binary search $v$ over $N(v_{j_i}, l)$;
            Push $v$ into $B_i$ once finding $v$ in $N(v_{j_i}, l)$;
    if $i == p$ then
        hash search over bitmap representation $B(u_i)$ ;
        Push $v$ to $B_p$ once finding the common vertex $v$;
return $B_p$;
```

called *L*abel-*P*artitioned *C*ompressed *S*parse *R*ow (LPCSR) (Definition 7), which leverages the burst I/O mode.

In WOJ-based subgraph matching algorithms, one frequent operation is to access a vertex v's neighbor list $N(v,l)$ whose neighbor vertex label is l. In the traditional CSR structure, all vertices' neighbors are consecutively arranged without label-based partitioning. Thus, to access $N(v,l)$, CSR needs more redundant memory access and filtering computation. To improve the efficiency of accessing $N(v,l)$, GSI [24] proposes a GPU-friendly data structure named Partitioned Compressed Sparse Row (PCSR). It partitions the data graph based on labels[2]. To address the non-consecutiveness of vertex IDs caused by graph partitioning, it hashes vertex IDs to a set of fixed-size hash buckets called *groups* and reorganizes CSR based on group IDs. When accessing $N(v,l)$, GSI computes the group ID that v is hashed to and uses a warp to search the group concurrently to locate $N(v,l)$. Due to hash conflicts, multiple groups may be probed to find the overflowed vertices.

PCSR [24] is not suitable for FPGA for two reasons. First, PCSR has the higher probability of group overflows on FPGA because the memory transaction bandwidth of FPGA is only 64B, but that of GPU is 128B, which means more groups may be fetched to locate $N(v,l)$. Second, the hash technique in PCSR cannot guarantee the access locality and is detrimental to burst I/O. For example, given two consecutive vertices v_1 and v_2, they may be hashed into groups far from each other. Thus, the system has to launch two separate read operations to fetch them rather than one burst read.

Therefore, to make full use of burst I/O for efficient access to $N(v,l)$, we propose the LPCSR structure (Definition 7). We divide a data graph G into a set of *neighbor label-partitioned graphs* $G(l)$ as follows:

Definition 6 (Neighbor Label-Partitioned Graph)**.** *Given a graph G and a vertex label l, the Neighbor Label-Partitioned Graph (denoted as $G(l)$) is a subgraph of G induced by all edges adjacent to at least one labeled-l vertex.*

An example of $G(B)$ that contains all edges adjacent to B-labeled vertices in G is given in Figure 5.

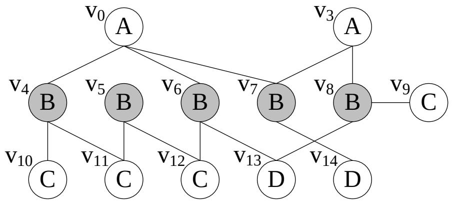


Fig. 5. a neighbor label-partitioned graph $G(B)$ of G in Figure 1(b)

An efficient data structure should support locating $N(v,l)$ in $O(1)$ time and read $N(v,l)$ in the linear time ($O(|N(v,l)|)$), which is the design goal of our proposed LPCSR. Given a data graph G, we partition it into k neighbor label-partitioned graphs $G(l_i)$, $i = 0, ..., k-1$. For each $G(l_i)$, we first build the traditional CSR R_i. Due to the non-consecutiveness of vertex IDs in the partitioned graph (e.g., no vertices v_1 and v_2 in $G(B)$ in Figure 6), we propose two extra levels in LPCSR.

Index List idx. The position of each vertex v_j in the *vertex array* of the CSR R_i corresponding to partitioned graph $G(l_i)$ if $v_j \in G(l_i)$, $j = 0, ..., |V|-1$, $i = 1, ..., k$, are collected sequentially to form an array, called *index list*, idx.

Example 5.1. Given the data graph G in Figure 1(b), we construct LPCSR in Figure 6. v_0, v_3 and v_{14} are three vertices in the *vertex array* of the CSR of $G(B)$. Their positions in the *vertex array* are 0, 1, and 7, respectively. For v_0 and v_3, because they do not have neighbors labeled A, we will store 0 and 1 in their first elements in idx, respectively. For v_{14}, because its first element in idx stores its position 9 in the *vertex array* of the CSR of $G(A)$, 7 will be stored in its second element.

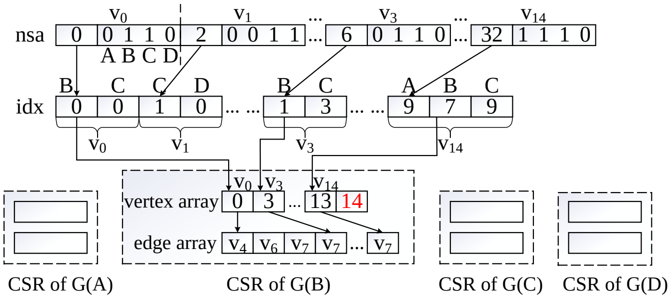


Fig. 6. LPCSR structure

Neighborhood Structure Array nsa. For each vertex v_j, $j = 0, ..., |V|-1$, we record v_j's starting offset in idx and existence bitmap $B(v_j)$. $B(v_j)$ has k bits and '1' in the i-th bit denotes $v_j \in G(l_i)$, $i = 0, ..., k-1$. All vertices' starting offsets and existence bitmaps are concatenated to form nsa.

Example 5.2. As shown in Figure 6, nsa has 15 elements, where each element corresponds to a vertex's neighborhood structure in Figure 1(b). For example, v_1' neighborhood structure consists of a 32-bit offset and a 4-bit bitmap. The offset

[2] [24] partitions the data graph based on *edge* labels, and the operation $N(v,l)$ is also based on edge labels.

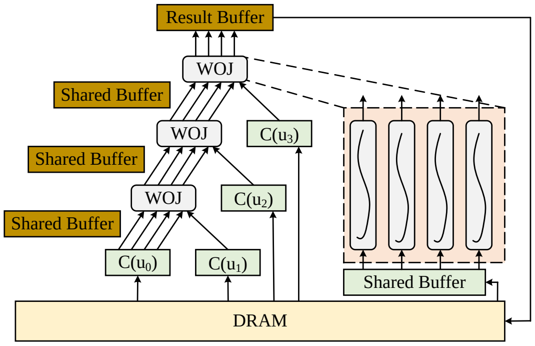


Fig. 7. Joint parallelized pipeline

records the starting position of v_1 in idx, which is 2. The bitmap is 0011, which means v_1 only exists in the *vertex arrays* of the CSRs of $G(C)$ and $G(D)$.

Definition 7 (LPCSR structure). *Given a data graph $G(V, E, L)$, LPCSR has four levels, including index list idx, neighborhood structure array nsa, vertex arrays, and edge arrays in the traditional CSRs.*

In LPCSR, the top two levels nsa and idx aim to rapidly locate a vertex v's neighbor list $N(v, l)$ with neighbor label l. If the l-th bit of v's bitmap in nsa is '1', we will get the position of v in the *vertex array* of the CSR of $G(l)$ from idx. The rest are multiple CSRs with different neighbor labels for accessing $N(v, l)$ consecutively and quickly.

Example 5.3. To illustrate how to locate a neighbor list in LPCSR, we give an example of fetching $N(v_{14}, B)$ in Figure 6. First, we read v_{14}'s corresponding element in nsa. The beginning index of v_{14} in idx is 32, and the existence bitmap is 1110. We check the 1-th bit of v_{14}'s bitmap is '1'. Then we count the number of 1 before this bit. In this case, the total number is 1. Thus, we read the 33-th (32+1) element in idx, which is 7. Finally, we read the 7-th element in $G(B)$'s *vertex array* and get the offset of the *edge array*, which is 13.

VI. Joint Parallelized Pipeline

To improve throughputs, a naive approach is to design multiple independent pipelines (Algorithm 1). However, multiple independent pipelines do not share the intermediate results while processing and may lead to workload imbalance. Thus, we propose a *joint parallelized pipeline* strategy for WOJ-based subgraph matching on FPGA, as shown in Figure 7. We implement *multiple-pipelines-in-one PE* where they push (or pull) intermediate results to (or from) *the same on-chip shared buffer*. Such a design addresses workload imbalance among multiple pipelines and provides more opportunities for memory coalescing since more neighbor lists are fetched from DRAM together. However, it also brings a new issue of writing conflicts when final results are written back to DRAM in parallel between different pipelines. To take optimization opportunities and avoid writing conflicts, we design a memory access coalescing mechanism to exploit burst I/O (discussed in Section VI-A). Also, we propose a space-saving pre-allocated write back strategy (discussed in Section VI-B).

A. Memory Access Coalescing

When adopting WOJ to match query vertices, most off-chip memory transactions come from the neighbor lists transfer from LPCSR on DRAM to BRAM. In the basic design, each independent pipeline fetches neighbor lists directly from DRAM. Although we can coalesce memory access inside each independent pipeline, the *joint parallelized pipelines* provide more opportunities to coalesce memory access. It is because multiple pipelines share common buffers, and more access requests are issued simultaneously.

1) Coalescing Mechanism: The memory controller will coalesce the memory access requests of multiple partial results from the partial result buffer with a coalescing width w, where w is limited by the buffer size. In our experiments, we set $w = 8$. Specifically, for those vertices with the same label l, we compute the offsets of their neighbor lists in parallel. Then, for these to-read neighbor lists, we measure the gaps among them. If the gap between two vertices' neighbor lists is below a threshold δ, we will combine the read operations of these two close neighbor lists by tolerating redundant data. The threshold δ is affected by the configuration of the FPGA card and can be measured experimentally, where it implies that the cost of the burst read to the two neighbor lists is equal to the cost of two random reads. After coalescing, the memory controller will get groups of burst read requests and launch burst read to fetch the data from DRAM into the on-chip shared buffer.

For example, as shown in Figure 8(a), there are three random memory access requests to $N(v_1, l)$, $N(v_3, l)$ and $N(v_5, l)$. We combine the three requests with two different lists (i.e., $N(v_2, l)$ and $N(v_4, l)$) to fetch them with a single run of *burst read* instead of three random accesses. In this case, $N(v_2, l)$ and $N(v_4, l)$ are redundant data.

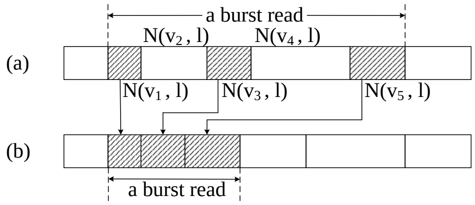


Fig. 8. A motivation example of graph reordering

2) Graph Reordering: We hope to coalesce more random memory requests while suffering from less redundant data. Since we store neighbor lists in vertex order, a different vertex order influences the access locality and the wasted read cost. To improve the efficiency of coalescing memory access on FPGA, we can find a suitable order for burst read in preprocessing phase based on the access locality and the wasted read cost between vertices. Figure 8 depicts the motivation of our graph reordering. After reordering, the wasted read of a burst read by combing the three memory requests is reduced significantly (=0) compared to before ordering. Although many graph reordering methods have been studied in previous works [48], [49], their contexts differ from ours. For example, Wei [49] exploits vertex locality to rearrange vertex order to reduce

CPU cache miss ratio in a CPU cacheline. It does not consider the waste read of accessing neighbor lists continuously.

Specifically, we re-order vertex IDs so that any two vertices v_i and v_j's neighbor lists are closer to each other in the *edge array* of the LPCSR if and only if (1) they are often accessed together during WOJ-based subgraph matching; and (2) the size of gaps between their neighbor lists should be minimized.

We first partition vertices based on vertex labels into different groups to find a good vertex order. Then, we reorder vertices in each group. Finally, vertices in different groups are concatenated to obtain the vertex order. We focus on reordering vertices with the same label in the following.

Definition 8 (Access Continuity Score). *Given a graph $G = (V, E, L)$ with k vertex labels, the* continuity score *of accessing vertices' neighbor lists in G is defined as follows:*

$$F(G, w, \pi) = \sum_{l \in L(G)} \sum_{v_i \in V(l)} \sum_{j=\max\{0, i+1-w\}}^{j=i-1} S(v_i, v_j) \quad (1)$$

where π is a vertex ID assignment function (i.e., $\pi(V) \rightarrow \{0, \ldots, |V|-1\}$), $L(G)$ denotes all vertex labels, $V(l)$ represents all vertices labeled l, w is the coalescing width and $S(v_i, v_j)$ is the compactness *between v_i and v_j, defined in Definition 9.*

Definition 9 (Vertex Compactness). *Given two vertices v_i and v_j ($j > i$), the compactness between them considers common neighbors and continuous access length from v_i's neighbor list to v_j's neighbor list in* LPCSR*, formally defined as follows:*

$$S(v_i, v_j) = \sum_{l \in L(G)} \frac{|N(v_i, l) \cap N(v_j, l)|}{\sum_{a=i}^{a=j} |N(v_a, l)|} \quad (2)$$

We demonstrate the vertex compactness of any two vertices in Figure 9. The numerator $|N(v_i, l) \cap N(v_j, l)|$ is the number of common neighbors labeled l between v_i and v_j. The denominator $\sum_{a=i}^{j} |N(v_a, l)|$ is the access length between $N(v_i, l)$ and $N(v_j, l)$, including needed read and wasted read.

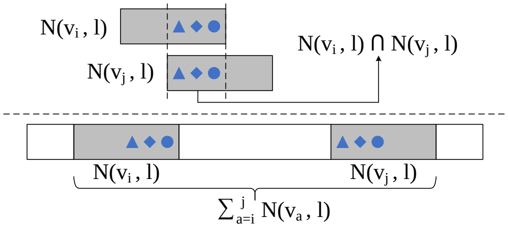


Fig. 9. Compute vertex compactness

Definition 10 (Graph Reordering Problem). *Given a graph $G = (V, E, L)$ and a coalescing width w, the graph reordering problem is to find a vertex ID assignment function $\pi(V) \rightarrow \{0, \ldots, |V|-1\}$ to maximize the access continuity score $F(G, w, \pi)$.*

Theorem 5.1 The graph reordering problem is NP-hard.

Proof. (sketch.) We prove this graph reordering problem is NP-hard by reducing it to a maximum traveling salesman problem without returning to the start vertex, similar to [49]. □

Heuristic Algorithm. Due to the NP-hardness of this problem, we propose a heuristic algorithm. Algorithm 3 shows the pseudo codes about finding the order over vertex sets $V(l)$ with one label l. We can repeat the algorithm on vertex sets with different labels and finally splice all vertices together because ordering vertices with different labels is mutually exclusive.

Algorithm 3: LPGO Algorithm

```
Input: data graph $G$ and the coalescing width $w$
Output: an order $\pi$ of vertices labeled with $l$ in $G$
/* $cmn$ records the number of common neighbors between
   $v_i$ and each $v_j$ added to $\pi$ before $v_i$ in the current size-$w$
   window */;
allocate $w \times |V(l)| \times |L|$ three-dimensional array $cmn$ ;
foreach $v \in V(l)$ do
    $Fscore(v) \leftarrow 0$, $vis[v] \leftarrow 0$;
select a start vertex $v_s$, $\pi[0] \leftarrow v_s$, $vis(v_s) \leftarrow 1$;
maintain a priority queue $q$ of length $w$;
for $i \leftarrow 1$ to $|V(l)| - 1$ do
    $v_e \leftarrow \pi[i-1]$;
    /* $v'$ and $v_e$ has one more common vertex $v$, thus,
       update the corresponding item in $cmn$ */;
    foreach $v \in N(v_e)$ do
        foreach $v' \in N(v, l)$ do
            if $vis[v'] == 0$ then
                $cmn[(i-1)\%(w-1)][v'][L(v)]$++;
    /* $Fscore(v') = \sum S(v_j, v')$, $v_j$ is added before $v'$
       within size-$w$ window*/;
    compute $Fscore(v')$;
    update the priority queue $q$ using $Fscore(v')$;
    $v_{max} \leftarrow q$.pop();
    $\pi[i] \leftarrow v_{max}$, $vis(v_{max}) \leftarrow 1$;
return $\pi$;
```

B. Space-Saving Pre-allocated Write Back Strategy

Let us consider the last WOJ step of the running example in Figure 7, which shows three partial matches corresponding to subquery $\mathcal{Q}_{[u_0,u_1,u_2]}$ and the last join vertex is u_3. Each pipeline extends one partial match m_i ($i = 1, 2, 3$) to obtain final matches. When they write these final matches to DRAM in parallel, write conflicts may occur. Other WOJ steps besides the last one in the pipeline (Figure 7) also have writing conflict issues due to a shared on-chip buffer. However, it is cheap to use *locking* strategy for BRAM and URAM, but it is costly to flush final results to DRAM by *locking*. To avoid write conflicts, one naive solution is using two-step output scheme like GpSM [23] and GunrockSM [22] on GPU. In the first step, each processor performs a complete join operation to count matches and calculate the output addresses based on the prefix-sum. In the second step, each processor performs the same join operation again and writes the results to the addresses in parallel. However, this method doubles the amount of work and thus suffers performance issues.

To avoid joining twice, GSI [24] proposes a *Prealloc-Combine* approach, which allocates the minimum length of the neighbor list of space for each partial match to avoid conflicts. Considering m_1, the allocated memory space for $m_1[v_9, v_8, v_3]$ is $min(|N(v_9, D)|, |N(v_8, D)|, |C(u_3)|)$, where $|N(v_9, D)|$ denotes the number of labeled-'D' neighbors of v_9 and $|C(u_3)|$

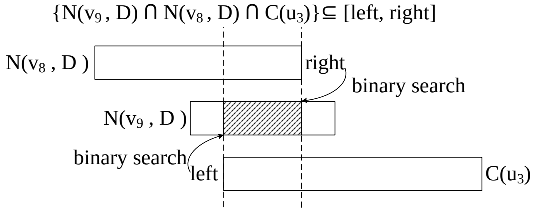


Fig. 10. Compute pre-allocated space

denotes the number of candidates for query vertex u_3. However, only 30% of the pre-allocated space is resultful in GSI.

We propose a space-saving pre-allocated strategy in Figure 10. To extend m_i, we intersect k sorted lists L_i $(i = 1, ..., k)$, such as extending m_1 by $N(v_9, D) \cap N(v_8, D) \cap C(u_3)$ in Figure 10. We first find $left$ and $right$, defined as follows.

$$left = max(min(x_i | x_i \in L_i), i = 1, ..., k)$$
$$right = min(max(x_i | x_i \in L_i), i = 1, ..., k)$$

Then, we count the number $\mathcal{N}$ of items whose values are between $left$ and $right$ at the shortest list L_i (i.e., the shaded area in Figure 10). Finally, the allocated space for m_i is $\mathcal{N}$. Our method always allocates less space than GSI for each pipeline. Experiments show our method can improve the space utilization up to 48% than GSI (see Table VI).

C. Optimization: FPGA-CPU Co-processing Strategy

Since our method performs the whole WOJ pipeline in FPGA, after transferring the input to the FPGA kernel, the CPU host is idle only to wait for returning final results. An FPGA-CPU co-processing solution can improve the overall performance. Due to the highly skewed degree distribution of real-world graphs (like power-law distribution), extending high-degree vertices will spend much more on-chip storage space and thus reduce the width of memory access coalescing. Fortunately, in a power-law distribution, there are only a few vertices with high degrees. Therefore, we can move several high-degree vertices to the CPU host.

Specifically, the CPU host divides the matching tasks into two parts based on each candidate's degree. The CPU will process a small number of high-degree candidates, and the rest of the low-degree candidates will be transferred to the FPGA kernel. The partition threshold is determined in the following.

First, given a join order O, we denote that the forward neighbors $N_+^O(u)$ of a query vertex u are u's neighboring query vertices that are matched after u. Given a candidate v of a query vertex u, we define its workload $W(v) = \sum_{u_i \in N_+^O(u)} \prod_{u_j \in N_+^O(u) \wedge u_j \leq u_i} |N(v, L(u_j))|$. Next, we partition candidates V into two parts V_C and V_F based on R, where R is the maximum size of the buffer on BRAM, $V_C = \{v_i | v_i \in V \wedge min(d_l(v_i)) > R\}$ and $V_F = \{v_i | v_i \in V \wedge min(d_l(v_i)) \leq R\}$. $min(d_l(v_i))$ is the minimum of the label-constrained degrees of v_i among different $G(l)$, which can be precalculated offline. Finally, we compute CPU's workload $W_C = \sum_{v_i \in V_C} W(v_i)$ and FPGA's workoad $W_F = \sum_{v_i \in V_F} W(v_i)$. If $W_C > tW_F$, finish. Otherwise, we ceaselessly move v_i with the maximum $min(d_l(v_i))$ from V_C to V_F. The parameter t is measured experimentally.

VII. Experiments

In this section, we evaluate the effectiveness of FASI and present the experimental results.

Setup. We have implemented FASI in C++ under Xilinx Vitis [3] 2020.1 development environment. Both the CPU host and the FPGA kernel are compiled by Vitis's built-in g++ based complier. We use a CentOS Linux server with two Intel Xeon Gold 5218 2.30GHz CPUs, 512GB host memory and one Xilinx Alveo U200 Data Center Accelerator Card for running CPU-based and FPGA-based systems. Meanwhile, the GPU-based systems run on another CentOS server with two NVIDIA Titan XP (3840 cuda cores and 12 GB global memory). The FPGA accelerator card is equipped with 64 GB off-chip DRAM, 35 MB on-chip memory (including 7MB BRAM and 28MB URAM), and 892,000 LUTs (Look-up Tables). It is attached to the CPU host through PCIe Gen 3.0 × 16. To avoid occasionality, when evaluating the elapsed time of each query, we run it 5 times and report the result using the average of the 5 runs.

TABLE II
Characteristics of datasets.

Dataset	$\|V\|$	$\|E\|$	$\|L\|$[a]	AD[b]	MD[c]	ACE[d]	Type[e]
patents	3.7M	16.5M	20	8.8	793	0.08	r
Youtube	1.1M	2.9M	25	5.3	28754	0.08	r
LiveJournal	4M	34.2M	30	17.3	14815	0.28	r
Orkut	3M	117.5M	24	76.2	33313	0.17	r
WatDiv	86M	549.2M	32	12.6	44.6M	0.10	s
Twitter	42M	1.5B	32	70.5	3.1M	0.07	r

[a-c] $|L|$, AD and MD denote the vertex label number, the average degree and the maximum degree, respectively.
[d-e] ACE and Type denote the average clustering coefficient and graph type (r:real-world and s:synthetic.)

Comparative Algorithms. For performance comparison, we evaluate our method with six state-of-the-art subgraph matching algorithms, including three CPU-based solutions CECI [38], DAF [42] and RAPID [43], two GPU-based solutions GpSM [23] and GSI [24], and a FPGA-based solution FAST [35]. To the best of our knowledge, there is only one former solution on FPGA for subgraph matching. For the CPU-based and GPU-based competitors, we choose the top fastest ones in our experiments. All the source codes come from the original authors and are also implemented in C++.

Datasets. We conduct our experiments on both real-world and synthetic datasets. The characteristics of these datasets are listed in Table II. We obtain two vertex-labeled graphs (patents and Youtube) from [46] and randomly assign vertex labels to other unlabeled graphs. Note that we only use the first five datasets in Table II except for the scalability experiments, since the basic version of our method (BASIC) and some comparative algorithms cannot work on billion-scale graphs.

Queries. To keep the query graphs for various datasets consistent, we randomly generate 50 unlabeled query graphs as query templates varying the vertex number from 3 to 7 and the edge number from 2 to 16, including star-like, line-like, clique-like, and hybrid. The maximum vertex number is

[3] https://www.xilinx.com/products/design-tools/vitis.html

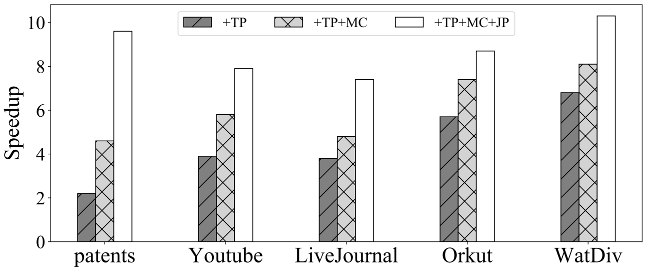


Fig. 11. Performance of optimization techniques

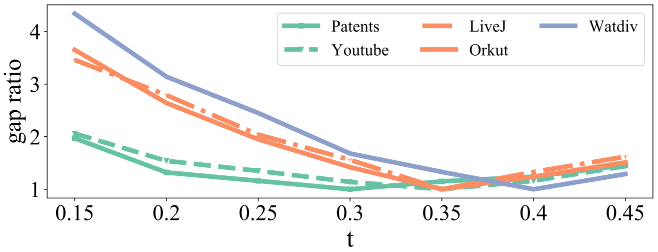


Fig. 12. The normalized time gap ratio varying t

consistent with FAST. Then, we randomly assign a label to each vertex for every unlabeled query graph.

Metrics. To evaluate the performance of an algorithm, we measure the elapsed time in milliseconds from receiving a query graph to outputting all results, which includes the CPU execution time and the hardware accelerator execution time (equal to 0 for CPU-based algorithms). To make each query terminate reasonably, we set a time limit of 10 minutes. We do not report the elapsed time if the query has timed out or is out-of-memory.

A. Resource Utilization of the FPGA kernel

We set the number of PEs to 8 in the FPGA kernel to maximize the on-chip resource utilization and the processing throughput of FPGA. Table III reports the resource utilization and the clock rate of our FPGA kernel. We can see that the utilization rate of BRAM and URAM is high, indicating our caching mechanism's effectiveness.

TABLE III
RESOURCE UTILIZATION OF THE FPGA KERNEL

Algorithm	LUT	Register	BRAM	URAM	Clock Rate
FASI	31.34%	14.87%	84.69%	59.24%	205MHz

B. Evaluating Optimizations in FASI

In this subsection, we evaluate three optimizations of FASI, including task partitioning (TP), memory access coalescing (MC), and joint parallelized pipelining (JP). We show their improvements in performance in Figure 11. We give FASI's basic implementation using LPCSR and independent pipelining without task partitioning and memory access coalescing, called Basic. Then, we compare the performance improvement of each optimization with the previous implementation by adding the three optimizations to Basic one by one and report speedup ratios over Basic. We denote the full-optimization version as FASI, used in the following experiments.

Effectiveness of Task Partitioning. We have evaluated the time gap between CPU and FPGA by varying t (A smaller gap means a higher overlap). The results in Figure 12 indicate $t = 0.35$ can achieve the best performance. After adding the task partitioning optimization, FASI outperforms Basic on every dataset by up to 6.8x (on Watdiv). The speed-up ratio varies among datasets due to different skewed distributions.

For example, we can see that the most prominent performance improvement is 6.8x on Watdiv, and the smallest is 2.2x on patents. The reason is that the degree distribution on Watdiv is skewed more than on patents. In BASIC, FPGA requires to pre-allocate much more on-chip memory to process higher degree vertices on Watdiv. Therefore, several very high-degree vertices can significantly impact the performance of BASIC on Watdiv. However, since the maximum vertex degree on patents is not very large, the performance of BASIC on patents does not become abysmal. Therefore, the performance improvement of BASIC+TP is more evident on Watdiv than patents. In other words, task partitioning is more effective for graphs with a highly skewed degree distribution.

Effectiveness of Memory Access Coalescing. In BASIC, there are many random off-chip memory accesses to fetch required neighbor lists. To speed it up, we apply a memory access coalescing mechanism, exploiting burst I/O on FPGA. The memory controller will coalesce the read requests of different vertices and transform them into burst reads to access multiple consecutive neighbor lists. According to Figure 11, the memory access coalescing optimization results in up to 2.0x improvements compared with Basic+TP. By tolerating redundant data and exploiting the burst read, our memory coalescing technique reduces the overhead of random memory access to off-chip DRAM and thus improves performance.

TABLE IV
AVERAGE COALESCING RATIO AND SPACE EFFICIENCY (BEFORE AND AFTER GRAPH RE-ORDERING).

Dataset	without graph reordering		LPGO	
	cr[a]	swr[b]	cr	swr
patents	35%	22%	52%	14%
Youtube	40%	26%	45%	18%
LiveJournal	29%	14%	54%	11%
Orkut	31%	21%	41%	13%
WatDiv	33%	19%	37%	17%

[a,b] cr: average coalescing rate; swr: space wasted rate.

We further demonstrate the efficiency of our graph reordering algorithm LPGO by two metrics: average coalescing rate (cr) and space wasted rate (swr). We obtain the coalescing rate by dividing the number of coalesced neighbor list accesses by the total number of neighbor list accesses. This ratio indicates the possibility of successful memory coalescing. The space wasted rate is calculated by dividing the size of redundant neighbor lists by the total size of coalesced read-in lists, which reflects the space efficiency of the proposed coalescing technique. Both metrics are attained from all queries on five datasets. After reordering, the coalescing rate improves significantly, especially on LiveJournal and Orkut, because of their higher average clustering coefficient (0.28 and 0.17). The space wasted rate also goes down.

TABLE V
AVERAGE COALESCING RATIO AND SPACE EFFICIENCY (INDEPENDENT PIPELINES VS. JOINED PARALLELIZED PIPELINES)

Dataset	IP		+JP	
	cr	swr	cr	swr
patents	52%	14%	64%	15%
Youtube	45%	18%	52%	19%
LiveJournal	54%	11%	58%	11%
Orkut	41%	13%	46%	16%
WatDiv	37%	17%	39%	17%

Effectiveness of Joint Parallelized Pipelining. Figure 11 shows that the joint parallelized pipelining optimization achieves a further speedup compared with BASIC+TP+MC (up to 2.1x on patents). This is because: (1) all the pipelines can get partial results as input in the shared buffer and thus have balanced workloads; (2) the shared partial result buffer provides more opportunities to coalesce memory accesses. We demonstrate the performance improvement of memory access coalescing using joint parallelized pipelining in Table V.

TABLE VI
THE TIME COST AND SPACE UTILIZATION USING TS, PC AND OUR STRATEGY ON DIFFERENT DATASETS.

Dataset	Time cost(ms)			Space Utilization		
	TS	PC	ours	TS	PC	ours
patents	28	18	20	100%	22%	54%
Youtube	15	10	11	100%	24%	42%
LiveJournal	459	292	302	100%	13%	61%
Orkut	1183	619	647	100%	15%	58%
WatDiv	12339	6706	6743	100%	18%	44%

In addition, when using joint parallelized pipelining, writing back the final results generated by different pipelines lead to conflicts. Thus, we propose a space-saving pre-allocation strategy to avoid that and compare it with the traditional two-step output scheme(TS) [22], [23] and Prealloc-Combine approach (PC) used in GSI [24]. We compare two metrics: the time cost and the space utilization (in Table VI). We can see that our strategy obtains up to 1.83x speedup than the two-step output scheme and up to 48% drop of the space cost than the Prealloc-Combine approach.

C. Evaluating LPCSR structure

To verify the efficiency of LPCSR, we compare it with the traditional CSR, and the GPU-friendly PCSR in GSI [24]. We implement them in FASI and evaluate the average elapsed time and the total space cost on multiple datasets in Table VII. LPCSR reduces the average elapsed time by 11.4x than CSR and 30% than PCSR. CSR without label-based partitioning performs worse than the other two since it has higher memory access cost and filtering cost. There are two reasons for the unsatisfactory effect of PCSR. One is that the maximum width of one memory transaction on FPGA is only half of the width of a global memory transaction on GPU, resulting in fewer elements in each group in PCSR and more groups to be accessed when locating a vertex's neighbor list. The other is that successive vertices are hashed to different groups, which lowers the possibility of coalescing memory access.

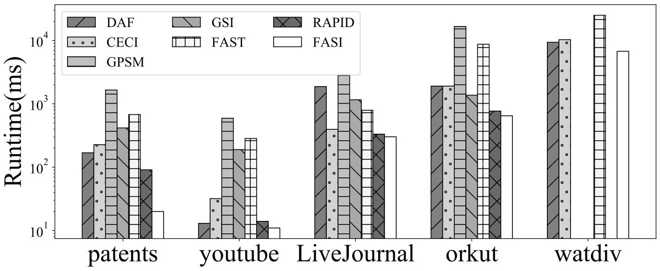


Fig. 13. Overall performance comparison of different algorithms

Furthermore, the space cost of LPCSR is also lower than PCSR because PCSR creates lots of groups in order to reduce the hash conflicts, and the space of many groups is wasted. If the number of groups is reduced to lower space cost, the hash conflicts will increase rapidly, and the performance of PCSR will worsen. Due to introducing two extra levels, LPCSR occupies more space than CSR, but it has much better time performance.

We also report the offline build time of LPCSR in Table VIII. We can see that it brings a certain amount of time overhead. However, it is worthwhile and affordable because the build time is a one-time expense and does not increase as the number of queries increases.

D. Comparing with Existing Algorithms

Figure 13 reports the average query runtime of FASI compared with existing algorithms CECI [38], DAF [42], RAPID [43], GpSM [24], GSI [23] and FAST [35]. Note that the results of GpSM and GSI on WatDiv are omitted because they are out of memory. RAPID also reports an error on WatDiv. As shown in Figure 13, FASI outperforms all comparative algorithms and achieves 14.5x average speedup.

Compared with CPU-based algorithms. FASI outperforms DAF by 4.21x on average (from 1.38x to 6.21x), CECI by 4.1x on average (from 1.31x to 11.35x) and RAPID by 2.02x on average (from 1.21x to 4.55x). The former two algorithms also adopt WOJ to compute matches, and RAPID is a hybrid of exploration-based and join-based methods. However, they are serial algorithms with no pipeline parallelization, so their performance is worse than our FPGA-based algorithm.

Compared with GPU-based algorithms. Our FASI outperforms GSI by 11.01x on average (from 2.11x to 20.85x) and

TABLE VII
AVERAGE ELAPSED TIME AND SAPCE COST OF FASI USING CSR, PCSR AND LPCSR ON DIFFERENT DATASETS.

Dataset	Time cost(ms)			Space cost(MB)		
	CSR	PCSR	LPCSR	CSR	PCSR	LPCSR
patents	229	32	20	161.6	487.2	206
Youtube	36	14	11	32	93.6	40.8
LiveJournal	1209	445	302	305.6	785.6	369.6
Orkut	1872	785	647	964	1902	1084
WatDiv	18465	7625	6743	5081.6	37417.6	9209

TABLE VIII
THE OFFLINE BUILD TIME OF LPCSR.

Dataset	patents	Youtube	LiveJournal	Orkut	WatDiv
Time(s)	11.43	3.32	18.25	43.79	315.23

GpSM by 42.9x on average (from 9.41x to 53.9x). Both GSI and GpSM adopt *materialized evaluation* in the whole WOJ processing. Each join step has to output all intermediate results to global memory on GPU and does not take the pipeline parallelism in FASI. The performance of GpSM is even worse because it adopts a two-step output scheme to avoid writing conflicts, which brings a large amount of redundant work. Furthermore, GPU-based algorithms GPSM and GSI often run out of memory when processing queries on large datasets or ones that produce lots of results (i.e., on WatDiv).

Compared with FPGA-based algorithms. FASI outperforms the only FPGA-based subgraph matching solution FAST by 15.91x on average (from 2.62x to 33.95x). Different from our method that the whole subgraph matching process is done within FPGA, FAST consumes a lot of CPU workload to build CST structure, which is quite time-consuming in FAST.

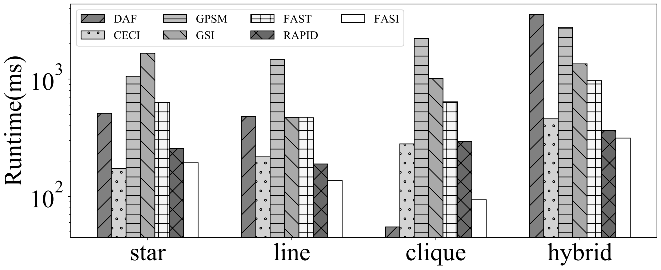
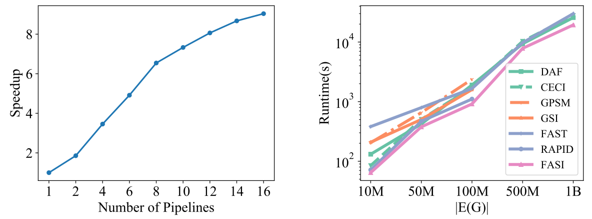


Fig. 14. Elapse time on LiveJournal of different types of queries

Evaluating Different Query Structures. For a more detailed comparison, we classify the queries based on their structures into four categories: star-like, line-like, clique-like, and hybrid. Star-like queries have only one vertex whose degree is more than one. Line-like queries feature vertices that form a line or a cycle. Clique-like queries are structurally similar to a clique. Each vertex in such queries has at least two neighbors. Hybrid queries are the combination of the former three query types. Using LiveJournal as an example, we show the specific runtime of four different types of queries.

As shown in Figure 14, FASI achieves a speedup of 4.1x on average in terms of star-like queries, thanks to our special treatment for one-degree vertices that avoids producing lots of intermediate results. In addition, compared with line-like queries, FASI performs better on clique-like queries. In other words, as the density of query graphs increases, the acceleration ratio of FASI also increases. Our pipeline design intended for WOJ plays a role here, as it fully utilizes the computing resource when dealing with the intersection of multiple lists. Such a gap is more expansive against FAST since there are more non-tree edges to be verified in FAST when queries are denser. Note that DAF can work better for clique-like queries because its failing set pruning strategy is more effective in dealing with this kind of query. Confronting a hybrid query, FASI splits it into a dense subgraph and a set of one-degree vertices for their respective processing. Therefore, we can credit FASI's advantage on hybrid queries(5.7x on average) to all the techniques above.



(a) Scalability with the number of pipelines on patents

(b) Scalability with different scales of Twitter

Fig. 15. Scalability test

E. Scalability

This subsection carries out the scalability test of FASI.

Varying the number of pipelines. We vary the number of pipelines from 2 to 16 and run all the queries on patents. Figure 15(a) shows the increasing trend of the average speedup as the number of pipelines increases. In general, the increase in the number of pipelines only reduces the speedup of the individual pipeline a little, thanks to our joint parallelized pipelining and memory access coalescing, which lowers the possibility of the limitation of multiple pipelines' memory access. Note that FPGA's on-chip memory is limited. Thus the number of pipelines is limited to FPGA's on-chip resources.

Varying $|E(G)|$. To test the impact of data size on query performance, we generate a series of datasets of different sizes using Twitter by randomly sampling a given number of edges (e.g., 10M, 50M, 100M, 500M, 1B). Figure 15(b) indicates the average elapsed time of different algorithms on datasets with different $|E(G)|$. Note that when the number of edges in the data graph is over 100M, GpSM and GSI fail due to GPU's global memory limitations. Besides, the runtime of FAST rises sharply as the data size grows larger because of the increasing CST scale. In contrast, the runtime of FASI rises much more slowly than others, which benefits from LPCSR and the space-saving pre-allocated write back strategy.

VIII. Conclusion

This paper proposes an FPGA-friendly subgraph matching algorithm (FASI), which utilizes FPGA's burst read and dataflow feature. Different from FAST that spends lots of time in building CST at query runtime and only uses FPGA for edge checking, FASI implements the whole WOJ-based subgraph matching algorithm in FPGA. FASI exploits FPGA's dataflow feature to optimize the pipeline join. With the help of LPCSR and the memory access coalescing, FASI can significantly reduce the costly random memory access between BRAM and DRAM on FPGA. Although the performance of FASI is reduced when dealing with data graphs with too many high-degree vertices, we mitigate this problem by moving a small number of high-degree vertices to the CPU host. Our experimental results on different datasets demonstrate that FASI outperforms other state-of-the-art algorithms significantly.

Acknowledgment

This work was supported by NSFC under grant 61932001 and U20A20174. Lei Zou is the corresponding author of this paper.

References

[1] T. A. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock, "New specifications for exponential random graph models," *Sociological methodology*, vol. 36, no. 1, pp. 99–153, 2006.

[2] J. Wang and J. Cheng, "Truss decomposition in massive networks," *arXiv preprint arXiv:1205.6693*, 2012.

[3] W. Fan, "Graph pattern matching revised for social network analysis," in *Proceedings of the 15th International Conference on Database Theory*, 2012, pp. 8–21.

[4] N. Pržulj, D. G. Corneil, and I. Jurisica, "Efficient estimation of graphlet frequency distributions in protein–protein interaction networks," *Bioinformatics*, vol. 22, no. 8, pp. 974–980, 2006.

[5] W. Kim, M. Li, J. Wang, and Y. Pan, "Biological network motif detection and evaluation," *BMC systems biology*, vol. 5, no. 3, pp. 1–13, 2011.

[6] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 335–346.

[7] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 425–440.

[8] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 716–727.

[9] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[10] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, "Taming subgraph isomorphism for rdf query processing," *arXiv preprint arXiv:1506.01973*, 2015.

[11] M. R. Garey, "A guide to the theory of np-completeness," *Computers and intractability*, 1979.

[12] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[13] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 405–418.

[14] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[15] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009, pp. 192–203.

[16] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[17] 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*, vol. 26, no. 10, pp. 1367–1372, 2004.

[18] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 337–348.

[19] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1199–1214.

[20] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[21] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 133–144, 2012.

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

[23] 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*. Springer, 2015, pp. 299–315.

[24] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: Gpu-friendly subgraph isomorphism," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1249–1260.

[25] B. Yang, K. Lu, Y.-h. Gao, X.-p. Wang, and K. Xu, "Gpu acceleration of subgraph isomorphism search in large scale graph," *Journal of Central South University*, vol. 22, no. 6, pp. 2238–2249, 2015.

[26] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1067–1082.

[27] 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.

[28] A. Rahman, J. Lee, and K. Choi, "Efficient fpga acceleration of convolutional neural networks using logical-3d compute array," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1393–1398.

[29] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 15–24.

[30] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–9.

[31] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated fpgas: Fpga framework for convolutional neural networks," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 265–268.

[32] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An fpga framework for edge-centric graph processing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 69–77.

[33] N. Engelhardt and H. K.-H. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[34] S. Zhou and V. K. Prasanna, "Accelerating graph analytics on cpu-fpga heterogeneous platform," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2017, pp. 137–144.

[35] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng, "Fast: Fpga-based subgraph matching on massive graphs," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1452–1463.

[36] C. Kankanamge, S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 1695–1698. [Online]. Available: https://doi.org/10.1145/3035918.3056445

[37] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, no. 4, pp. 20:1–20:44, 2017. [Online]. Available: https://doi.org/10.1145/3129246

[38] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1447–1462.

[39] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 260–269.

[40] Y. E. Ioannidis and Y. C. Kang, "Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization," in *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, 1991, pp. 168–177.

[41] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *arXiv preprint arXiv:1903.02076*, 2019.

[42] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1429–1446.

[43] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: a holistic approach to subgraph query processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 176–188, 2020.

[44] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi, "Efficient subgraph matching using gpus," in *Australasian Database Conference*. Springer, 2014, pp. 74–85.

[45] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons learned from exploring the backtracking paradigm on the gpu," in *European Conference on Parallel Processing*. Springer, 2011, pp. 425–437.

[46] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1083–1098.

[47] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: new developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, pp. 5–16, 2013. [Online]. Available: https://doi.org/10.1145/2590989.2590991

[48] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using simd instructions," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1587–1602.

[49] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1813–1828.