POLITECNICO DI TORINO
Repository ISTITUZIONALE

Low-Power Subgraph Isomorphism at the Edge Using FPGAs

Original
Low-Power Subgraph Isomorphism at the Edge Using FPGAs / Bosio, R., Brignone, G., Urso, T., Lazarescu, M.T., Lavagno, L., Pasini, P.. - In: IEEE ACCESS. - ISSN 2169-3536. - 13:(2025), pp. 67127-67135. [10.1109/ACCESS.2025.3560405]

Availability:
This version is available at: 11583/2999761 since: 2025-05-02T10:12:06Z

Publisher:
IEEE

Published
DOI:10.1109/ACCESS.2025.3560405

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

18 November 2026

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 February 2025, accepted 5 April 2025, date of publication 14 April 2025, date of current version 23 April 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3560405

RESEARCH ARTICLE

Low-Power Subgraph Isomorphism at the Edge Using FPGAs

ROBERTO BOSIO, (Graduate Student Member, IEEE), GIOVANNI BRIGNONE, (Graduate Student Member, IEEE), TEODORO URSO, (Graduate Student Member, IEEE), MIHAI T. LAZARESCU, (Senior Member, IEEE), LUCIANO LAVAGNO, (Life Senior Member, IEEE), AND PAOLO PASINI
Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Roberto Bosio (roberto_bosio@polito.it)

This work was supported in part by the Key Digital Technologies Joint Undertaking through the REBECCA Project, receiving support from the European Union, Greece, Germany, Netherlands, Spain, Italy, Sweden, Türkiye, Lithuania, and Switzerland, under Grant 101097224; in part by the Spoke 1 on Future HPC of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4–Next Generation EU; and in part by the European Union-Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP E13C22001870001, Partnership on "Telecommunications of the Future" (Program "RESTART"), under Grant PE00000001.

ABSTRACT Subgraph matching is a significant problem in several fields, including like social network analysis, chemical compound search, and fraud detection. While current solutions using CPU, graphics processing units (GPUs), and data center field-programmable gate arrays (FPGAs) deliver high performance, they consume significant power, making them unsuitable for resource-constrained edge environments. This article introduces a novel low-power FPGA accelerator that enables efficient subgraph matching using a small FPGA-based accelerator through three main innovations: a flexible two-level hash table architecture that minimizes memory accesses, a caching mechanism that reduces on-chip memory requirements, and a dynamic first-in first-out (FIFO) queue that efficiently buffers partial results. Experimental results on five real-world graphs show that the accelerator reduces energy consumption by an average of 75 × compared to state-of-the-art CPU solutions, and 107 × compared to GPU solutions, demonstrating that complex graph operations can be performed efficiently using even a small accelerator implemented on a reconfigurable platform.

INDEX TERMS Graph theory, FPGA, high-level synthesis, cache, query.

I. INTRODUCTION

Subgraph isomorphism (or subgraph matching) [2] is an NP-hard problem that searches for all copies of a given pattern (the query graph) within a larger and more complex data graph. For example, given the query graph $\{u_0, u_1, u_2\}$ and the data graph in Fig. 1, the (labeled undirected) subgraph isomorphism finds the two dashed subgraphs $\{v_2, v_1, v_4\}$ and $\{v_{10}, v_8, v_6\}$.

Subgraph isomorphism is used in various domains where graph patterns provide valuable information, like social network analysis [3], chemical compound search [4], and fraud detection [5].

Given the relevance and complexity of the problem, the literature offers numerous solutions that explore various computing architectures. CPU implementations [2], [6] typically adopt a recursive approach, evaluating one potential solution at a time. Advanced approaches incorporate complex pruning rules and efficient matching order to trim the search space.

GPUs are also widely used to solve subgraph isomorphism problems [7], [8]. GPUs offer significant computational power through their parallel processing capabilities. However,

[0]The associate editor coordinating the review of this manuscript and approving it for publication was Sun-Yuan Hsieh.

© 2025 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

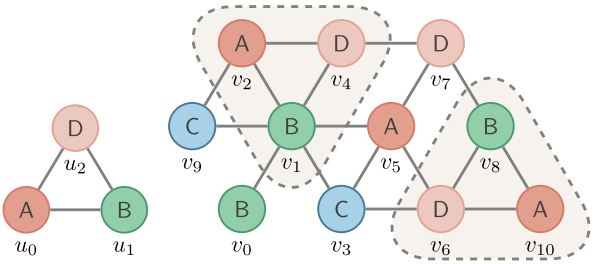


FIGURE 1. Query (a) and data graph (b) example, with the matches highlighted by triangles.

fully exploiting their parallelism is challenging due to warp divergence, where threads enter distinct execution paths, leading to inefficiencies.

Recent solutions target heterogeneous platforms with CPUs and FPGAs, adopting a hardware/software co-design paradigm [9], [10]. These approaches exploit datacenter-class FPGAs to accelerate the matching task, offloading part of the workload to high performance CPU.

State-of-the-art (SOTA) subgraph isomorphism research is primarily aimed at increasing throughput, given the growing importance of energy efficiency, especially in database applications [11]. On the other hand, we propose a novel approach for computationally efficient operations on relatively small FPGAs that can be used either for edge devices or in small reconfigurable accelerator boards.

Our approach[1] differs from previous implementations since: 1) it does not need cooperation with a power-hungry CPU, as the FPGA also accelerates the preprocessing task, and 2) it requires a limited amount of resources thanks to an efficient caching mechanism. The proposed methodology is implemented on a low-power embedded FPGA, a device typically not associated with such applications due to its limited resources. Experiments on real-world graphs show that our architecture consumes less energy per query than leading CPU solutions such as RapidMatch [12] by a factor of 8, DAF [6] by a factor of 75, and the GPU algorithm GSI [13] by a factor of 107.

II. BACKGROUND

In line with previous work [6], [12], we focus on connected undirected vertex-labeled graphs, but all the techniques presented are easily generalizable to cases using directed or edge-labeled graphs. Furthermore, the proposed approach also covers the case of unlabeled graphs as a special case where only one label exists.

A. PROBLEM STATEMENT

A graph $G(V, E)$ is a *set of vertices* V, a *set of edges* $E \subseteq V \times V$ and a labeling function $L : V \to \Sigma$, which assigns a label, from the *set of labels* Σ, to each vertex.

[1]A preliminary version of this work was presented as extended abstract in [1]

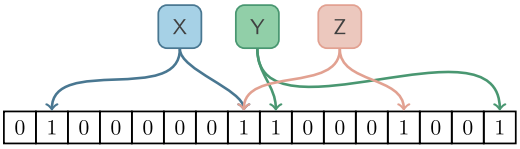


FIGURE 2. Insertion of three items in an empty Bloom filter with the number of hash functions $k = 2$ and the size of the bit array $m = 16$.

An edge $e(u, u') \in E$ connects two vertices $u, u' \in V$. The neighborhood $N(u)$ of a vertex $u \in V$ is the set of vertices connected to the vertex u by an edge of the graph and is formally defined as $N(u) = \{u' | u' \in V \wedge e(u, u') \in E\}$. The neighborhood subset of a vertex $N(u)$, containing only the vertices with label l, is $N(u, l) = \{u' | u' \in N(u) \wedge L(u') = l\}$.

Definition 1: Given a query graph Q and a data graph G, a subgraph isomorphism is an injective function $f : V(Q) \to V(G)$ such that $\forall e(u_1, u_2) \in E(Q) \implies e(f(u_1), f(u_2)) \in E(G), \quad \forall u \in V(Q) : L(u) = L(f(u))$.

Each injective function is a solution, which we will also refer to as a *match* or an *embedding*. A partial solution is an embedding that covers only a subset of $V(Q)$. The sequence in which the subgraph matching algorithm analyzes the vertices, named the matching order ϕ, is a permutation of the set of nodes $V(Q)$. The subgraph matching problem consists in finding all subgraphs of G that are isomorphic to Q.

B. JOIN-BASED SUBGRAPH MATCHING

A subgraph query can be resolved by a series of join operations on a relational database [14], [15] that combine elements from two or more sets based on a common attribute to produce a unified result. Each edge in the query can be reinterpreted as a relation, informally defined as a table storing a set of edges. Given an edge $e(u_i, u_j) \in E(Q)$, the relation $R(u_i, u_j)$ contains only the data graph edges connecting vertices with labels $L(u_i)$ and $L(u_j)$. The evaluation of the subgraph query involves joining all relations.

Example 1: Given the triangle query in Fig. 1 (a), $R(u_1, u_2, u_3) = R(u_1, u_2) \bowtie R(u_1, u_3)$ finds all the triplets that form an open triangle in the data graph in Fig. 1 (b) and $R(u_1, u_2, u_3) \bowtie R(u_2, u_3)$ removes all tuples that do not contain the closing edge, thus finding all the matches.

Pairwise join plans are a viable solution for evaluating subgraph queries. In graph terms, each binary join extends the set of partial solutions by introducing an edge. Unlike traditional pairwise approaches, our method for joining uses multiple set intersections to extend each solution by a single vertex, as explained in Section IV.

C. BLOOM FILTERS

Bloom filters are a commonly used data structure that provides a space-efficient means of representing sets [16]. A Bloom filter is characterized by the size of its bit array, denoted as m, and the number of hash functions, denoted as k, that map each element to a specific bit. The process of inserting an element into the set involves setting the values of k bits to 1, with the hash functions determining the positions of

these bits, as shown in Fig. 2. To verify the existence of an element, the k bits generated by the hash functions are checked. Although false positives are possible, false negatives are not. The Bloom filter representation allows an approximate set intersection to be computed by a bitwise AND operation, and the cardinality of a set is approximately equal to the number of bits at 1 divided by k.

We use these properties to significantly reduce the number of costly external memory accesses (see Section IV).

III. RELATED WORK

The literature presents several methods to address this relevant problem, including optimized software, specialized hardware such as application-specific integrated circuits (ASICs), and more. In this study, we focus on standalone solutions using off-the-shelf hardware and exclude distributed solutions from our scope.

A. CPU IMPLEMENTATIONS

The first proposed algorithm for efficient subgraph isomorphism resolution sequentially matches query nodes with vertices in the data graph using a backtracking approach [2]. Subsequent research focused mainly on minimizing the exploration space [17], [18]. DAF [6] proposes a complete data structure that replaces the data graph as a search space and supports an enumeration phase based on set intersection instead of edge verification. VEQ [19] exploits equivalences to prune redundant computations, and RapidMatch [12] integrates a join-based strategy with the preprocess and filter approach.

The CPU remains the most versatile and scalable option because it can implement complex pruning rules and heuristics that account for the diverse characteristics of graphs.

B. GPU IMPLEMENTATIONS

GpSM [20] uses a pairwise join plan to find matches, taking advantage of GPU parallelism by computing matches according to a breadth-first search (BFS) strategy. Later, GSI [13] outperformed previous work by adopting a vertex-oriented join based on set intersection. Since a BFS approach requires extensive storage for intermediate results, EGSM [7] addresses the issue by using a hybrid BFS-DFS solution that can switch to the depth-first search (DFS) approach as memory becomes scarce. On the other hand, STMatch [8] opts for a DFS approach and mitigates workload imbalances by introducing work-stealing techniques.

GPU solutions can achieve higher throughput in certain cases, but consume significantly more power than CPU solutions.

C. FPGA IMPLEMENTATIONS

FAST [9] is the first subgraph matching accelerator implemented on an FPGA. It preprocesses graphs on a CPU to construct the candidate search tree (CST) data structure, which stores candidate subgraphs. The CST is then partitioned such that each partition fits into the on-chip memory of the FPGA.

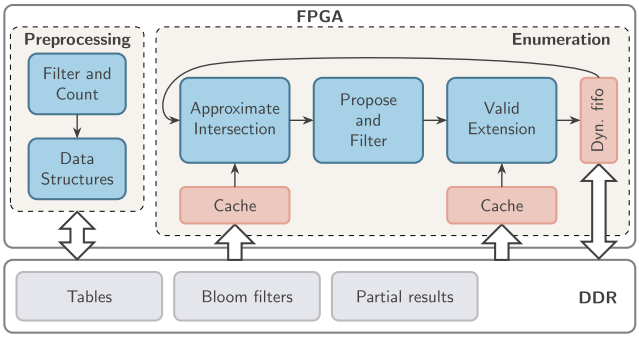


FIGURE 3. Architecture of the proposed system, split in *preprocessing* to read the graph and assemble the data structures, and *enumeration* to identify the matches using intersections.

The enumeration step based on edge verification is repeated for each partition of the CST.

FASI [10] proposes a worst-case optimal join (WCOJ) approach, with an offline CPU pre-processing step. It avoids flushing intermediate results into the off-chip memory by storing them to the FPGA on-chip memory.

Both algorithms defer some of the workload to the CPU to improve overall performance, but this significantly increases power consumption.

IV. ARCHITECTURE

Figure 3 shows the architecture of the proposed system, which operates in two main phases, both implemented on FPGA: *preprocessing* to read the graph and assemble the data structures, and *enumeration* to identify the matches through a series of set intersections.

A. PREPROCESSING

Figure 4 shows the storage arrangement of each relation $R(u_i, u_j)$, organized into two data structures: the *hash table*, a matrix that stores memory addresses, and a collection of *adjacency lists*, logically partitioned into segments and accessed based on the values retrieved from the *hash table*.

The preprocessing phase constructs the *hash tables* and incorporates spatial locality, which is exploited by the subsequent caching mechanism. It is important to note that our preprocessing approach differs from the conventional methods used in algorithms designed for GPUs and CPU, which focus on advanced strategies to eliminate as many invalid candidates as possible before starting the enumeration phase. Instead, our goal is to minimize external memory accesses by maximizing the hit ratio of the caching mechanism.

The initial phase organizes the graph edges according to their $R(u_i, u_j)$ relations, discarding those connected to vertices without labels matching the query, which cannot contribute to solutions. This process splits each adjacency list $N(v_i)$, grouping neighborhood nodes based on their label, as shown in the first step of Fig. 4.

The second stage builds a hash table for memory addresses using a matrix with $H1 = 2^{h_1}$ buckets (a blue row in Fig. 4). Each bucket points to a superset of an adjacency list. Due to

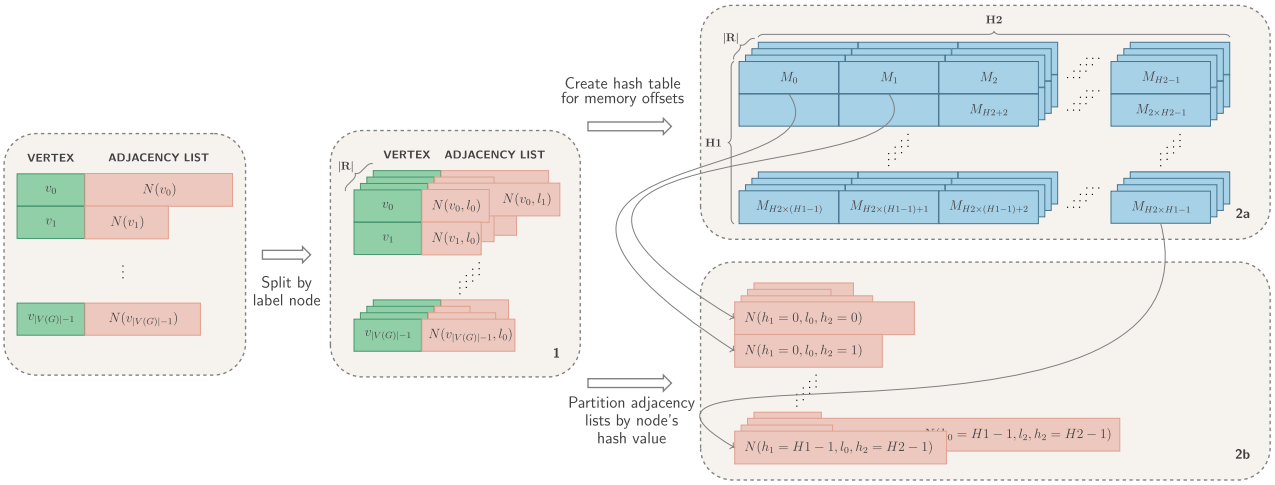


FIGURE 4. Data structures aimed at optimizing algorithm execution. The initial collection of adjacency lists is partitioned based on the neighbor's label (1). Each relation is then preprocessed to obtain a hash table storing $H1 = 2^{h_1}$ rows (2a), each mapping to $H2 = 2^{h_2}$ memory addresses (M_i), and the array of partitioned adjacency lists (2b), accessed using M_i as memory offset.

hash collisions, multiple adjacency lists may be stored in the same bucket. To retrieve an adjacency list $N(v_i, l)$, the hash value of v_i is truncated to h_1 bits, and the resulting value is used to access the appropriate bucket in the hash table, which returns the memory address where the adjacency list is stored. The set of nodes retrieved may contain more elements than the original adjacency list, but the enumeration phase ensures the correctness of the final result.

The second parameter h_2 works similarly to h_1, dividing each bucket into $H2 = 2^{h_2}$ sub-backets (individual blue cells). Each cell stores the memory offset of a portion of an adjacency list, partitioned based on the hash value of the vertices.

Example 2: Procedure to determine if edge $e_0 = E(v_0, v_1)$ exists: access the hash matrix table containing pointers to adjacency lists with label $L(v_1)$; find the row corresponding to $h_1(v_0)$ and the column corresponding to $h_2(v_1)$; use the pointer from that position to read all edges in the subset looking for e_0.

The construction of the data structure is done in two successive steps (Fig. 4). First, the graph is read and the number of collisions in each hash table cell is determined, allowing the memory address for the start of each partitioned adjacency list to be calculated. The rearranged data is then stored in memory similar to a counting sort [21], with each adjacency list sorted based on the first h_2 bits of the vertex hash values. At the same time, Bloom filters are generated and associated with a hash table bucket representing their vertices.

The proposed method extends the partitioned compressed sparse row (PCSR) representation introduced in GSI [13]. Unlike PCSR's single-level approach, our method organizes data on two levels. The first level of hashing still allows access to a node's neighborhood, while the second level allows for fast membership checking, i.e., verifying the presence of a vertex in an adjacency list, without requiring a complete read-through. Also unlike PCSR, our method uses non-fixed bucket sizes due to the inclusion of a collision counting step, a flexibility that helps alleviate overflow problems and improves data locality.

The memory consumption of off-chip data structures is directly proportional to the number of query edges, since each edge can potentially become a relation. This memory usage grows exponentially with the sum of h_1 and h_2. Increasing the hash table parameters reduces hash collisions, but results in increased off-chip memory usage. Conversely, smaller values for the hash table parameters save memory but increase the number of slow and energy-intensive off-chip memory transactions during the enumeration phase.

B. ENUMERATION

To incorporate an additional query vertex into an existing partial match, we used a method based on set intersection, a technique often used in previous studies [6]. Specifically, given a partial solution and a query vertex u_i to cover, all possible extensions are obtained by selecting the query vertices already mapped in the neighborhood of u_i and then intersecting their mapping adjacency lists.

Example 3: Considering the query in Fig. 1, with a match order $\phi = (u_0, u_1, u_2)$ and a partial match $P = (v_{10}, v_8)$, the candidates for u_2 are found intersecting the neighborhood of v_{10} restricted to $L(u_2)$, i.e. $N(v_{10}, D) = \{v_6\}$ with $N(v_8, D) = \{v_6, v_7\}$, which gives $\{v_6\}$, a correct mapping for u_2, and the final match $\{v_{10}, v_8, v_6\}$.

Algorithm 1 shows the workflow of the enumeration phase, which reads, expands, and rewrites partial matches until there are no more partial solutions to expand. Our technique for computing the intersection is based on finding the smallest of the sets involved, and then probing its elements against the hash tables of the other sets. The process begins by reading a partial match and determining the next query vertex to match after ϕ. It then retrieves the Bloom filters representing the adjacency lists involved in the current set intersection. The filters are used to 1) identify the minimum set among the ones under consideration and 2) compute a new Bloom filter B

Algorithm 1 Enumeration

Input: Matching order ϕ, query Q, relations R
Output: Matches $\mathcal{M}$

```
1  F ← candidates for root φ[0];
2  while F ≠ ∅ do
3      P ← F.pop(), E_f ← ∅;
4      B ← ApproximateIntersection(P);
5      E ← ProposeAndFilter(R, P, B);
6      foreach v_i ∈ E do
7          if ValidExtension(R, P, v_i) then
8              E_f ← E_f ∪ {v_i};
9      if |P| == |Q(V)| − 1 then
10         M.push(P × E_f);
11     else
12         F.push(P × E_f);
```

representing the approximate intersection (Line 4). Then, the smallest adjacency list involved in the intersection is read from memory and filtered based on B, with the remaining elements stored in the extension set E (Line 5). The vertices in the extension set are then probed against the other adjacency list hash tables, which perform the full set intersection and remove erroneous nodes introduced by hash collisions (Lines 6 to 6). Finally, the last step generates a new solution for each valid extension found by adding the new node to the previous match (Lines 9 to 11).

C. MEMORY MANAGEMENT

Modern FPGA boards have a large off-chip memory (e.g., dynamic random access memory (DRAM)) with low bandwidth and high latency, and several small on-chip memories (e.g., flip-flops (FFs), lookup table (LUT) memories, block random access memories (BRAMs), and ultra random access memories (URAMs)) with high bandwidth and low latency. Designers explicitly transfer data between the different memories, possibly aided by caches [22].

Subgraph matching, like many graph algorithms, is memory-bound, so optimizing memory access is essential for good performance. This is done using high-locality data structures accessed via a cache [22] and a dynamic FIFO buffer to efficiently store partial results without saturating on-chip memory.

1) CACHE AND LOCALITY

Our architecture exploits spatial and temporal locality of memory references through a cache for FPGAs [22]. Spatial locality derives from the reorganizations of the adjacency lists and it is mainly exploited during the validity checks of an extension set. The preprocessing loosely sorts the adjacency lists on the first h_2 bits of the nodes hash values. The enumeration uses these lists as extension sets, and validates each node by a series of memory reads. Our hash table model linearly maps hash values to memory addresses, resulting in sequential accesses. Consequently, the cache loads to on-chip memory blocks of data that also contain the elements accessed in the following algorithm iterations. Therefore, it pre-fetches data from off-chip memory, hiding its latency.

Temporal locality comes from the BFS and is used between consecutive extensions of partial matches. Extensions are proposed based on set intersection, with sets coming from the adjacency list of nodes already matched in the partial solution. If two consecutive solutions share part of the mapping nodes, they share also part of the sets involved in the intersection. Our method prioritizes data reuse by preserving the order of partial solutions, thereby decreasing the number of distinct nodes between successive solutions.

2) RESULT FIFO

The algorithm uses a BFS approach, which produces a potentially large number of partial solutions that may not fit in on-chip memory. To address this challenge, we designed a dynamic FIFO queue that behaves like a standard on-chip FIFO when the number of results fits into the on-chip buffer. In contrast, when the data exceeds a predefined threshold, the dynamic FIFO stores partial matches in off-chip memory while maintaining a full on-chip buffer to mask memory latency. When the data falls below the threshold, the dynamic FIFO reverts back to storing data exclusively in on-chip memory.

We assume that data fits to off-chip memory, since FPGA accelerators normally do not use virtual memory.

D. HEURISTICS AND QUERY VERTEX ORDER

Before starting the kernel, the CPU selects a matching order of query vertices. This order has a significant impact on enumeration performance, because selecting less frequent query vertices earlier reduces the search space. In the literature, various heuristics have been proposed to select an optimal order without increasing the runtime excessively, many of which rely on information from the datagraph. The proposed method instead uses a lighter approach that relies solely on the structure of the query, which has proven to be effective [15].

The starting point for the matching order is ϕ, the query vertex with the highest degree. Then vertices are selected iteratively, based on the number of neighbors in ϕ. In case of a tie, we choose the node with the highest degree. The chosen order has the dual advantage of generating connected sequences, which avoids expensive cross-products between adjacency lists, and evaluating intersections between neighborhoods of already matched nodes as early as possible, effectively exploiting approximate intersection filtering.

As discussed in Section IV-A, the size of the hash table significantly impacts the enumeration runtime. The CPU heuristically computes two parameters, h_1 and h_2, based on the size of the data graph. These parameters, which determine the number of bits used to address the hash tables, exhibit an exponential relationship with the hash table size:

$$M \propto 2^{(h_1+h_2)} \tag{1}$$

where M represents the memory occupation. Therefore, to ensure a linear relationship between the graph size and the hash table size, these parameters must scale linearly with the logarithm of the number of edges in the data graph. This can be approximated as:

$$h_1 = a_1 + b_1 \log(|E|)$$
$$h_2 = a_2 + b_2 \log(|E|)$$

where the coefficients were determined empirically by interpolating data points that relate the number of edges in the graph to the hash table parameters that yielded the lowest runtimes.

V. RESULTS

We quantitatively compared our architecture with the fastest open source solutions for unordered vertex-labeled subgraph matching, namely RM [12], DAF [6], and GSI [13]. Our analysis omits EGSM [8] and STMatch [7] since their open-source implementations output incorrect results (we reported the issue to their GitHub repositories).

A. SETUP

The accelerator was described in C++ using Xilinx *Vitis HLS* 2023.2 and deployed on an AMD Kria KV260 operating at 290 MHz with 4 GB of off-chip memory and 3 MB of on-chip memory. CPU solutions were executed on a workstation running CentOS 7, equipped with a 16-core Intel i7-6900K CPU (3.2 GHz) and 128 GB of host memory. GPU solutions were executed on an NVIDIA GeForce GTX 1070 (1.5 GHz) with 1920 CUDA cores and 8 GB of global memory. Both the GPU and FPGA use 16 nm technology, while the CPU uses a 14 nm technology. The time elapsed from the start of the algorithm execution until all matches were found, excluding the time spent loading the data graph to memory from disk, was measured. The power consumption of the various devices were measured using software tools, `perf` for the CPU, NVIDIA `nvprof` for the GPU and `xmutil` for the FPGA. Queries were considered solved only if their execution time was lower than 10 minutes for the CPU and GPU, and 30 minutes for the proposed architecture, given the difference in clock frequency. DAF and RM execute on the CPU, with the former supporting multithreading, while GSI runs on the GPU.

B. DATASET AND QUERIES

Following previous studies, we use five real-world data graphs from SNAP [23]. For each data graph, we randomly assigned labels to the vertices from a small set to avoid trivial queries, as a larger number of labels would significantly reduce the search space. The properties of the graphs are reported in Table 1, which includes the number of vertices, edges, and labels, respectively $|V|$, $|E|$, and $|\Sigma|$, as well as the average and maximum vertex degree. Are generated 30 random query templates, ranging from 3 to 8 vertices and from 3 to 26 edges. This extends the complexity beyond what previous FPGA studies had considered [9], [10], as they had limited the size to a maximum of 16 edges. Table 2 reports resource usage for the board used to run the set or experiments being discussed.

TABLE 1. Dataset characteristics: number of vertices $|V|$, edges $|E|$, labels $|\Sigma|$, and vertex degrees: average d_{avg} and max d_{max}.

Datatset	$\|\mathbf{V}\|$ (k)	$\|\mathbf{E}\|$ (k)	$\|\mathbf{\Sigma}\|$	$\mathbf{d}_{\mathrm{avg}}$	$\mathbf{d}_{\mathrm{max}}$ (k)
Enron	37	184	5	10.0	1
GitHub	38	289	5	15.0	9
Gowalla	197	950	5	9.7	14
Dblp	317	1050	5	6.6	0.3
Wikitalk	2394	4660	5	3.9	100

TABLE 2. Resources used by the proposed accelerator: lookup table (LUT), flip-flop (FF), block random access memory (BRAM), digital signal processor (DSP), and ultra random access memory (URAM).

Board	LUT	FF	BRAM	DSP	URAM
Kria KV260	39814 (34 %)	67120 (29 %)	83.5 (58 %)	11 (1 %)	34 (53 %)

C. COMPARISON

In the tests, the proposed accelerator systematically outperforms SOTA implementations in terms of energy efficiency, achieving up to 13.6 × lower average energy consumption compared to RM (over Enron), up to 103 × compared to DAF (over Dblp), and up to 387 × compared to GSI (over Gowalla). As shown in Fig. 5, the proposed solution scales efficiently from simple to complex queries (from left to right), with the energy gap remaining approximately the same on a logarithmic scale, regardless of query complexity. This demonstrates the effectiveness of the optimizations, including the dedicated data cache mechanism, across a wide range of query sizes. Note that in the Wikitalk dataset, only the first nine queries were solved within the time threshold, as shown in Section V-C, and queries 28 and 29 are not included in Section V-C, for the same reason. In contrast, GSI struggles with many queries due to its large memory footprint and solves only 52 % of our test cases, primarily on smaller datasets.

FAST [9] and FASI [10] were not included, since neither the source code nor the bitstream are publicly available. However, [10] quantitatively compared the execution time of FASI with RM and FAST. FASI is on average 16 × faster than FAST and only 2 × faster than RM, despite using the combined power of a high performance CPU and a Xilinx Alveo U200 FPGA, which alone consumes up to 225 W. Although a direct evaluation of FAST and FASI is not possible, our measured improvements over RM suggest that the overall efficiency of our accelerator is superior to both.

To evaluate the efficiency of the caching mechanism, a kernel version without caches was implemented and its runtime compared on the same benchmarks, with the performance gains provided by the cache reported in Fig. 6. The box plots show that the effectiveness of the caching mechanism grows with the size of the datagraphs (see Table 1), since the increased number of memory reads to find all the matches makes the memory bottleneck more severe.

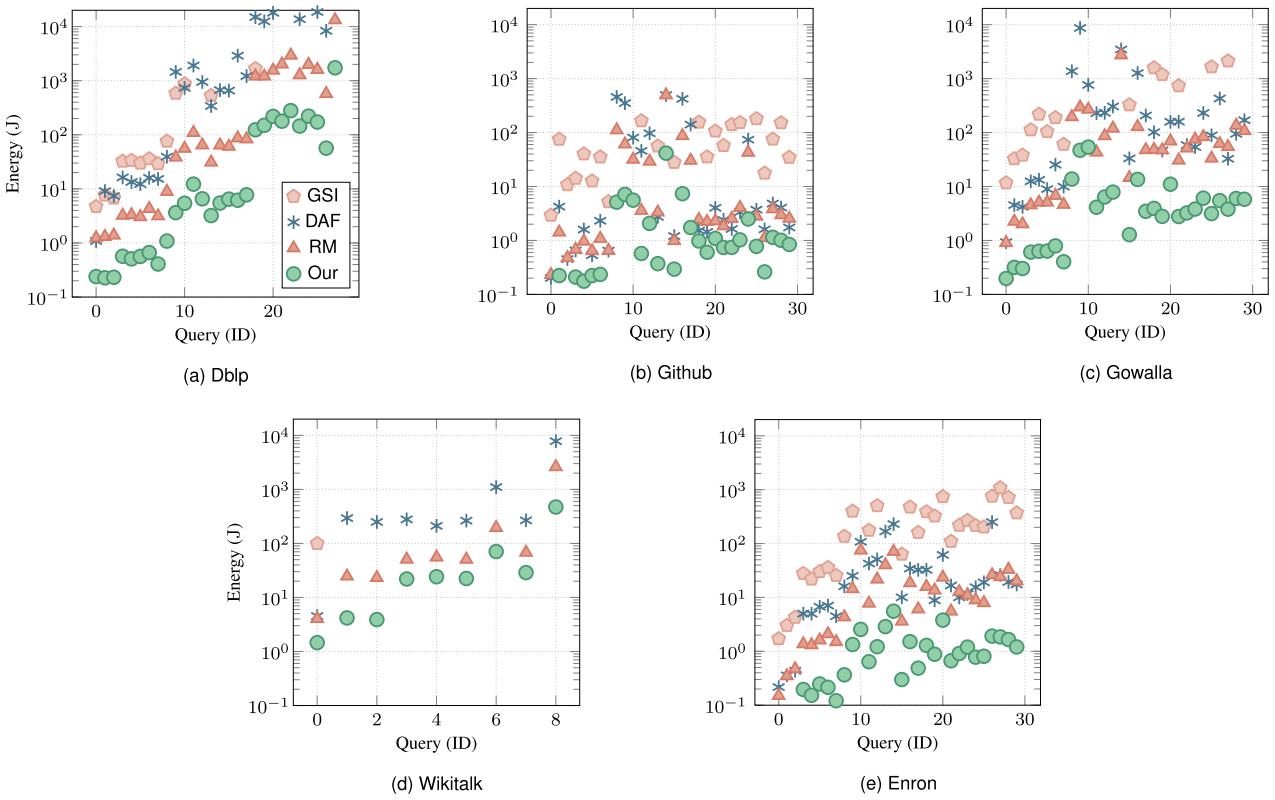


FIGURE 5. Energy consumption per query search for different datasets (the lower the better).

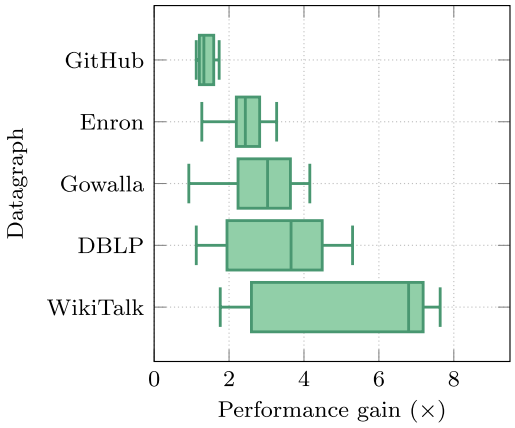


FIGURE 6. Execution time speedup thanks to caching compared to directly accessing off-chip memory.

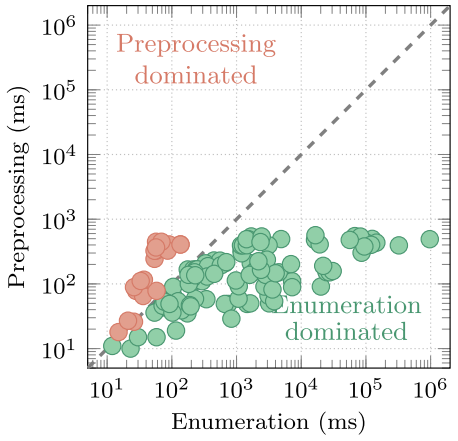


FIGURE 7. Preprocessing time for the proposed system rarely dominates the enumeration time.

The size of the hash table has a significant impact on kernel runtime (see Section IV-D), affecting both the preprocessing and enumeration phases. Increasing the dimensions of the hash table can minimize collisions and thus increase the efficiency of the enumeration step, but at the expense of longer preprocessing phase for simple queries. Figure 7 shows the distribution of query execution times (on the x-axis) versus preprocessing times (on the y-axis). The figure shows how our hash table sizing heuristics scale effectively, ensuring that the preprocessing time rarely dominates the overall execution, even for trivial queries.

Although this work primarily focuses on energy efficiency and our evaluation platform is significantly less powerful than those used by other approaches, our solution delivers competitive performance. Our average performance ranges from 7 × faster on average than GSI (running on GPU) to 1.6 × slower on average than RM (running on CPU), while consuming 36 × and 20 × less power, respectively.

Figure 8 shows the effectiveness of the proposed solution, by plotting power consumption versus average execution time for each architecture across all datagraphs. In this power-time space, oblique lines indicate iso-energy levels. This

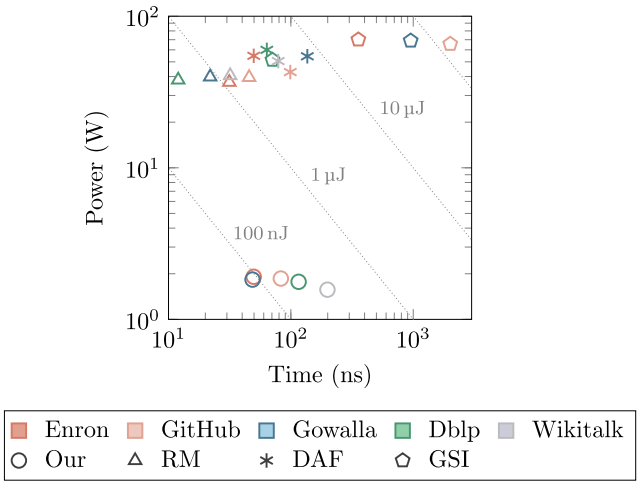


FIGURE 8. Average execution time per solution versus power consumption across various architectures, represented by the shapes, and datagraphs, represented by the colors. The proposed method is Pareto-optimal, achieving the lowest energy consumption.

visualization clearly shows that our solution is Pareto optimal across all datasets.

VI. CONCLUSION

We introduce a low-power FPGA subgraph isomorphism accelerator that shifts the focus from traditional raw performance to energy efficiency in graph pattern matching. Unlike prior solutions that rely on power-hungry data center hardware, our approach demonstrates the viability of efficient subgraph matching even on resource-constrained edge devices. The challenge of limited on-chip memory in embedded FPGAs has been effectively addressed through innovative data structures and an efficient caching mechanism, enabling memory-bound algorithms like subgraph matching to be processed without compromising performance.

Experimental validation on real-world graphs shows marked energy efficiency improvements, averaging 75 × better than CPU solutions and 107 × better than GPU solutions. These results are especially significant because they demonstrate that complex graph operations can be performed efficiently using a relatively small accelerator on a reconfigurable platform. Our work successfully fills the gap in energy-efficient subgraph matching, and its methodologies are applicable to other memory-bound algorithms, supporting the growing demand for low-power computing.

REFERENCES

[1] R. Bosio, G. Brignone, F. Minnella, M. U. Jamal, and L. Lavagno, "LESS: Low-power energy-efficient subgraph isomorphism on FPGA," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2024, pp. 1–2.
[2] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.
[3] W. Fan, "Graph pattern matching revised for social network analysis," in *Proc. 15th Int. Conf. Database Theory*, Mar. 2012, pp. 8–21, doi: 10.1145/2274576.2274578.
[4] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, New York, NY, USA, 2004, pp. 335–346, doi: 10.1145/1007568.1007607.
[5] A. Bodaghi and B. Teimourpour, "Automobile insurance fraud detection using social network analysis," in *Applications of Data Management and Analysis*. Cham, Switzerland: Springer, 2018, pp. 11–16, doi: 10.1007/978-3-319-95810-1_2.
[6] 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 *Proc. Int. Conf. Manage. Data*, Jun. 2019, pp. 1429–1446, doi: 10.1145/3299869.3319880.
[7] X. Sun and Q. Luo, "Efficient GPU-accelerated subgraph matching," *Proc. ACM Manage. Data*, vol. 1, no. 2, pp. 1–26, Jun. 2023.
[8] Y. Wei and P. Jiang, "Stmatch: Accelerating graph pattern matching on gpu with stack-based loop optimizations," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2022, pp. 1–13.
[9] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng, "FAST: FPGA-based subgraph matching on massive graphs," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, Apr. 2021, pp. 1452–1463.
[10] X. Su, Y. Lin, and L. Zou, "FASI: FPGA-friendly subgraph isomorphism on massive graphs," in *Proc. IEEE 39th Int. Conf. Data Eng. (ICDE)*, Apr. 2023, pp. 2099–2112.
[11] B. Guo, J. Yu, D. Yang, H. Leng, and B. Liao, "Energy-efficient database systems: A systematic survey," *ACM Comput. Surv.*, vol. 55, no. 6, pp. 1–53, Jul. 2023.
[12] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "RapidMatch: A holistic approach to subgraph query processing," *Proc. VLDB Endowment*, vol. 14, no. 2, pp. 176–188, Oct. 2020.
[13] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "GSI: GPU-friendly subgraph isomorphism," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 1249–1260.
[14] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, "EmptyHeaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, pp. 1–44, Jan. 2015.
[15] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 1083–1098, doi: 10.1145/3318464.3380581.
[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[17] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing Cartesian products," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1199–1214, doi: 10.1145/2882903.2915236.
[18] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. data*, Jun. 2008, pp. 405–418, doi: 10.1145/1376616.1376660.
[19] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Versatile equivalences: Speeding up subgraph query processing and subgraph matching," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2021, pp. 925–937, doi: 10.1145/3448016.3457265.
[20] H.-N. Tran, J. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Database Systems for Advanced Applications*, Cham, Switzerland: Springer, pp. 299–315, Jan. 2015.
[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., Cambridge, MA, USA: MIT Press, 2009.
[22] G. Brignone, M. Usman Jamal, M. T. Lazarescu, and L. Lavagno, "Array-specific dataflow caches for high-level synthesis of memory-intensive algorithms on FPGAs," *IEEE Access*, vol. 10, pp. 118858–118877, 2022.
[23] J. Leskovec and A. Krevl. (2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. [Online]. Available: http://snap.stanford.edu/data



ROBERTO BOSIO (Graduate Student Member, IEEE) received the master's degree from the Politecnico di Torino, in 2022, where he is currently pursuing the Ph.D. degree. His main research interests include high-level synthesis and dataflow architectures.



GIOVANNI BRIGNONE (Graduate Student Member, IEEE) received the M.S. degree in computer engineering from the Politecnico di Torino, Italy, in 2021, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. Luciano Lavagno. His research interests include high-level synthesis, digital hardware design, and HW/SW co-design.



TEODORO URSO (Graduate Student Member, IEEE) received the M.Sc. degree in mechatronic engineering from the Politecnico di Torino, Italy, in 2022, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications. His research interests include low-power low-energy acceleration of machine learning applications for high performance computing and edge devices.



MIHAI T. LAZARESCU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications from the Politecnico di Torino, Italy in 1998. He was a Senior Engineer with Cadence Design Systems and founded several startups. He is currently an Assistant Professor with the Politecnico di Torino. He co-authored over 60 scientific publications, four books, and international patents. His research interests include design tools for WSN/IoT platforms, ubiquitous environmental sensing, efficient neural networks, indoor human localization, edge and leaf IoT data processing, high-level HW/SW co-design, and synthesis.



LUCIANO LAVAGNO (Life Senior Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from UC Berkeley, in 1992. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He was an Architect with the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect with the Cadence CtoSilicon high-level synthesis tool. He co-authored four books and over 200 scientific articles. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.



PAOLO PASINI received the Ph.D. degree in computer and control engineering from the Politecnico di Torino, in 2017. He is currently a Postdoctoral Researcher with the Department of Electronics and Telecommunications (DET), Politecnico di Torino. He is active in the field of bit-level hardware model checking.