GraphMatch: Subgraph Query Processing on Steroids

JONAS DANN*, ETH Zürich, Switzerland
TOBIAS GÖTZ, TU Munich & SAP, Germany
DANIEL RITTER, SAP, Germany
JANA GICEVA, TU Munich, Germany
HOLGER FRÖNING, Heidelberg University, Germany
GUSTAVO ALONSO, ETH Zürich, Switzerland

Recently, graphs are becoming increasingly interesting in the context of large language models and as overlays for commercial databases. Subgraph query processing is an especially challenging workload for graph analysis that is bottlenecked by slow set intersection performance on CPUs. Previous work has shown the viability of utilizing hardware acceleration for related domains like graph and relational join processing.

We propose GraphMatch, a hardware-accelerated subgraph query processing system based on worst-case optimal joins (WCOJ). For efficient processing of various data and query graphs, we propose a novel set intersection algorithm, called MaxStep, that leverages hardware parallelism. GraphMatch combines MaxStep operators in a data flow architecture which efficiently solves multi-set intersections in subgraph query processing, superior to CPU-based approaches. GraphMatch achieves an average speedup of over 6.98× and 17.08×, compared to the state-of-the-art WCOJ-based systems GraphFlow and RapidMatch, respectively. On labeled graphs, GraphMatch outperforms the fastest subgraph query processing accelerator FAST by orders of magnitude.

CCS Concepts: • **Hardware** → **Emerging technologies**; • **Information systems** → *Data management systems*; • **Computer systems organization** → **Architectures**.

Additional Key Words and Phrases: Subgraph query processing; Set intersection; Hardware specialization; FPGA; Graph database

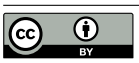
ACM Reference Format:
Jonas Dann, Tobias Götz, Daniel Ritter, Jana Giceva, Holger Fröning, and Gustavo Alonso. 2025. GraphMatch: Subgraph Query Processing on Steroids. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 332 (December 2025), 26 pages. https://doi.org/10.1145/3769797

1 Introduction

Graph-structured data is becoming increasingly interesting in many application areas [62], like social network analysis [64] and protein interaction network analysis [57], and more recently in the context of large language models (LLMs) [31] and as overlays for commercial database management systems. For example, graphs have been used in vector search for retrieval augmentation [40] and to improve reasoning capabilities [12] of LLMs, making graph overlays (e. g., AWS Neptune [9], SAP HANA Property Graph [52, 63], Microsoft Cosmos DB for Apache Gremlin [10]) a necessity to

*Work mostly done at SAP.

Authors' Contact Information: Jonas Dann, jonas.dann@inf.ethz.ch, ETH Zürich, Zürich, Switzerland; Tobias Götz, goetzt@in.tum.de, TU Munich & SAP, Munich, Germany; Daniel Ritter, daniel.ritter@sap.com, SAP, Walldorf (Baden), Germany; Jana Giceva, jana.giceva@in.tum.de, TU Munich, Munich, Germany; Holger Fröning, holger.froening@ziti.uni-heidelberg.de, Heidelberg University, Heidelberg, Germany; Gustavo Alonso, alonso@ethz.ch, ETH Zürich, Zürich, Switzerland.



ACM 2836-6573/2025/12-ART332
https://doi.org/10.1145/3769797

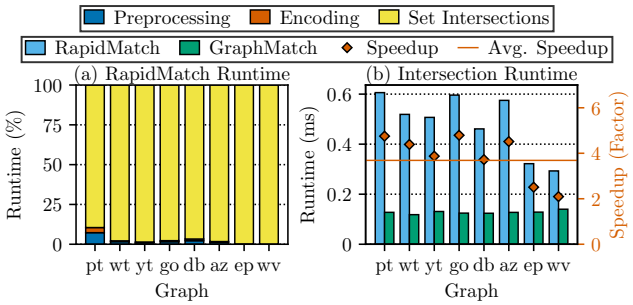


Fig. 1. RapidMatch (CPU) runtime breakdown (left), and RapidMatch and GraphMatch (FPGA) intersection operator runtime (right).

model unique views on the underlying data. Subgraph query processing is an especially challenging workload for analysis of such graphs where all embeddings identical to a query graph are found in a given data graph. In related work, there are two main approaches to this workload: backtracking- [16, 70] and join-based subgraph query processing [2, 53] that employ worst-case optimal joins (WCOJs). While the approaches are the same complexity-wise [67], join-based approaches are the current canonical approach. However, similar to many other workloads, subgraph query processing is limited by the stagnating performance scaling of current CPU hardware [30].

Hardware specialization has become the leading strategies to overcome this limitation [1]. For data management systems, there has been a large body of work that, for instance, covers relational data processing on GPUs [18, 43, 44, 54, 74] and field-programmable gate arrays (FPGAs) [47]. At the same time, hardware accelerators are becoming widely available in major cloud data centers [11, 58]. Notably, Microsoft has been deploying two FPGAs as network-attached accelerators per server in their Azure deployments [19]. Fully custom hardware architectures in the form of application-specific integrated circuits (ASICs) and FPGAs are particularly versatile for hardware specialization and provide massive data and pipeline parallelism (e. g., in related domains like graph processing [26] and JSON parsing [29]). However, a recent survey on non-relational data processing [27] identified a gap for hardware-accelerated subgraph query processing (there is only one hybrid CPU-FPGA backtracking-based system FAST [42]).

We propose GraphMatch, a hardware-accelerated, WCOJ-based subgraph query processing system. We observe that set intersections are the dominant operation on CPUs (cf. Figure 1(a)) for join-based subgraph query processing approaches (e. g., RapidMatch [67]) despite vectorized processing. Thus, we propose the novel MaxStep hardware set intersection algorithm that is specifically designed to leverage the massive data and pipeline parallelism of the underlying hardware by trading off work optimality for performance scaling. The potential of this hardware specialization approach for set intersection is shown in Figure 1(b), which compares RapidMatch using SIMD-based set intersection (cf. [36]) to MaxStep with an average speedup of 3.69× based on 5000 intersections of random neighborhoods. On top of that, GraphMatch implements a data flow architecture for subgraph query processing as a pipeline of MaxStep operators that keeps partial results on the chip and thereby saves expensive roundtrips to off-chip memory. With our GraphMatch query parser, we can configure this pipeline for arbitrary subgraph queries on-the-fly during runtime.

With this work, we make the following contributions:

- We propose and analyze the novel hardware set intersection algorithm MaxStep. (Section 3)

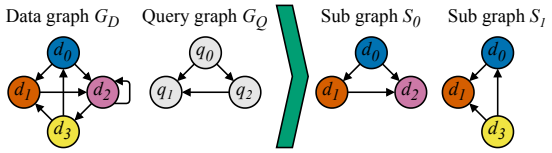


Fig. 2. Query example with subgraph isomorphisms.

- We design a hardware-accelerated subgraph query processing system based on WCOJs, called GraphMatch. (Section 4.1)
- We make GraphMatch configurable for dynamic queries and labeled graphs. (Section 4.2)
- We propose four performance optimizations for GraphMatch. (Section 4.3)

The resulting GraphMatch system shows performance scalability with an average speedup of 6.98× over GraphFlow [53] and 17.08× over RapidMatch [67] (Section 5). We also compare GraphMatch performance with accelerated systems on the labeled graph LDBC social network benchmark. Overall, we conjecture that hardware acceleration is well suited to solve the set intersection bottleneck of CPU-based subgraph query processing systems. However, we still see areas of improvements, for instance, for large query graphs, work balancing, and highly degree-skewed graphs (Section 6).

2 Background and Related Work

In this section, we discuss the two main approaches to subgraph query processing (i. e., backtracking- and join-based), hardware design principles focusing on clock frequency, and position GraphMatch in the context of the related work.

2.1 Subgraph Query Processing

Graphs are abstract data structures ($G = (V, E)$) comprising of a vertex set V and an edge set $E \subseteq V \times V$. Figure 2 shows an example of an unlabeled subgraph query processing task for directed graphs, given a data graph G_D with four vertices and seven edges, and an input query graph G_Q in the form of a triangle. The task computes either the homomorphisms or isomorphisms of the query graph within the data graph [67]. Both denote subgraphs of the same shape as the query graph, but homomorphisms allow duplicate vertices within subgraphs, while isomorphisms do not [67]. In the given example, we identify all isomorphisms of G_Q in G_D. Thus, all results of the task are triangular subgraphs of G_D, where edges of the same direction exist in the data graph.

Backtracking-based Subgraph Query Processing In a first step, backtracking explores the data graph and creates candidate sets of appropriate vertices for each query vertex. The second step takes those sets and enumerates all valid isomorphisms. In the literature, three different enumeration variations are known [67]: direct-, index-, and preprocessing-enumeration. They differ in the way they create the candidate sets (e. g., enumerate subgraphs directly or create an index structure on the data graph beforehand).

Join-based Subgraph Query Processing Worst-case optimal joins (WCOJ), as shown in Algorithm 1, guarantee a runtime complexity in the worst-case output cardinality of the algorithm [32, 55, 67]. Join-based subgraph query processing is based on the WCOJ algorithm Generic Join [33, 53, 55, 56, 67] that follows an iterative approach to construct all homomorphisms by joining one vertex at a time to a temporary query subgraph (partial matching). To find valid join candidates, it intersects the corresponding neighborhood sets of all previously matched vertices that share an edge with the to-be-added vertex in the query graph. All elements of the result set are joined to the current subgraph. This process creates multiple new extended subgraphs for

Table 1. Subgraph query processing systems, design principles, and properties.

Name	Platform	Approach	Key data structure	General	Parallel	Dir.	Lab.	Hom.	Iso.
CFLMatch [16]	CPU	Backtracking	Compact path index	☑	☐	☐	☑	☐	☑
DAF [35]	CPU	Backtracking	Candidate space	☑	☑	☐	☑	☐	☑
CECI [15]	CPU	Backtracking	Compact embedding cluster index	☑	☑	☑	☑	☐	☑
EmptyHeaded [2]	CPU	Compiled WCOJ	Trie	☐	☑	☑	☑	☑	☐
GraphFlow [53]	CPU	WCOJ & binary joins	Adjacency lists	☑	☑	☑	☑	☑	☐
RapidMatch [67]	CPU	WCOJ & hash joins	Encoded trie	☑	☐	☐	☑	☑	☑
GraphZero [51]	CPU	Compiled nested loops	Adjacency lists	☐	☐	☑	☑	☐	☑
GpSM [69]	GPU	Compiled joins	Compressed sparse row	☐	☑	☑	☑	☐	☑
GSI [75]	GPU	WCOJ	Partitioned compressed sparse row	☑	☑	☐	☑	☐	☑
GunrockSM [73]	GPU	Backtracking	Compressed sparse row	☑	☑	☐	☑	☐	☑
FAST [42]	FPGA & CPU	Backtracking	Candidate search tree	☑	☐	☑	☑	☐	☑
GraphMatch	FPGA	WCOJ	Compressed sparse row	☑	☑	☑	☑	☑	☑

Dir.: Directed graphs; Lab.: Labeled graphs; Hom.: Subgraph homomorphisms; Iso.: Subgraph isomorphisms; ☑: yes; ☐: no

Algorithm 1 Worst-case optimal join algorithm.

Given: Query graph $Q = (V_Q, E_Q)$, Data graph $D = (V_D, E_D)$

```
1: function WCOJ(i, M) // i: Index query vertex; M: Partial matching
2:     if i ≤ |V_Q| then
3:         C ← {}
4:         for all w ∈ ∩{N_D(m_j ∈ M) | u_j ∈ N_Q(v_i ∈ V_Q)} do
5:             C ← C ∪ WCOJ(i + 1, M ∪ w)
6:         return C
7:     return {M}
8: for all v ∈ V_D do // Run WCOJ for every vertex in the data graph
9:     WCOJ(1, {v})
```

the next iteration. The algorithm terminates after all of the subgraphs represent a valid matching or no subgraph can be extended anymore. A variation of Generic Join is LeapFrog Triejoin [71] that is optimal in the number of comparisons during set intersection. To compute isomorphisms with WCOJs, matchings where one data graph vertex matches multiple query vertices have to be filtered out during the join process. However, note that this may require more work and may impact worst-case optimality compared to a native subgraph isomorphism algorithm.

2.2 Hardware Design Principles

FPGAs map custom digital circuit designs (a set of logic gates and their connections) to a grid of resources (e. g., look-up tables and registers) connected with a programmable interconnection network. Synthesizing a design for such an architecture involves assigning (*place*) the logic gates to their physical equivalents on the chip and allocating interconnect wires for their connections (*route*). Contrary to instruction-based processors (CPUs and GPUs), the performance of a hardware design is not limited by the number of instructions that need to be executed to complete a workload but by (i) how efficiently the available hardware resources can be utilized to exploit parallelism and (ii) the achievable clock frequency of the design. The achievable clock frequency specifically is restricted by the longest path (measured in nanoseconds) between two registers which we call the critical path. The critical path needs to fit inside one clock cycle (e. g., 1/250MHz = 4ns). The path length is given by the sum of the time that electricity needs to travel through the wires between gates on the path and the time that the gates on the path need to stabilize their output after a signal arrives on the input. In complex hardware designs, the number of logic gates in the paths and the fan-out of signals become limiting factors to clock frequency. Less often, the general density of the

design becomes the limiting factor for routing because it restricts the freedom in the placement of the logic. To optimize clock frequency, we try to either restructure the hardware design altogether or pull apart the logic with more pipelining which, however, is not always possible (e. g., if there are tight feedback loops for backpressure). The performance of the design is linear in the clock frequency that the synthesis achieves.

2.3 Related Work

Computing subgraph iso- and homomorphisms is studied in detail in related work [48, 66, 76]. The result of the increased attention in research is a variety of approaches and systems. Table 1 depicts an overview of the current state-of-the-art and sets it in relation to our system GraphMatch. We also briefly discuss graph mining and graph database related work which we exclude from deeper investigation due to a different set of design considerations.

Backtracking-based Systems CFLMatch [16] and DAF [35] are backtracking-based systems that use a custom candidate data structures for subgraph query processing. CFLMatch decomposes the query graph into a core, forest, and leaves that are matched in that order due to their decreasing selectivity. DAF uses a candidate space data structure with dynamic programming, an adaptive query vertex ordering, and failing set pruning. CECI [15] splits up the data graph into embedding clusters to parallelize execution and employs pruning to reduce the number of intermediate matchings.

WCOJ-based Systems EmptyHeaded [2] introduced the WCOJ-based approach to subgraph query processing as a compilation-based system. It uses a trie data structure as an index to support subgraph queries on directed graphs. GraphFlow [53] extends EmptyHeaded's approach by combining it with binary joins with a query optimizer allowing for query plans which combine partial embeddings. These systems don't have subgraph isomorphism checks and thus only return subgraph homomorphisms. RapidMatch [67] is the most recent subgraph query processing system that proves that the backtracking and WCOJ approaches are equal complexity-wise. Based on this observation, it combines a join-based approach with backtracking-like candidate pruning. GraphZero, in turn, [51] is a compilation-based approach that tries to eliminate redundant computations in a nested loop structure.

Subgraph Query Processing Accelerators GpSM [69] uses two steps where they join once to determine the cardinality of the output in order to compute the output addresses, and then join again to obtain the results. GSI [75] divides the graph into partitions for each edge label which can lead to pointer arrays becoming large if one value has to be stored per vertices. To solve this, they introduce a hashing layer that maps vertex identifiers to pointers which costs one more memory request and hashing runtime. GunrockSM [73] uses GPU parallelism by exploring BFS-based, which trades off parallelism for more intermediate results that are oftentimes the bottleneck. FAST [42] introduces a subgraph query processing system for hybrid FPGA-CPU architectures. It computes a candidate search tree on the CPU that is loaded onto the FPGA for matching enumeration with computation overlapping.

Graph Mining Systems Graph mining [20–23, 34, 68] is a related domain to subgraph query processing. However, graph mining works on different assumptions that result in significantly different systems. In graph mining, systems are optimized to identify whole classes of queries (e.g., clique finding and k-motif counting) which are oftentimes not materialized but only counted. Additionally, graphs are almost always assumed to be undirected which restricts the expressiveness of queries. Nonetheless, the concepts behind graph mining have parallels in subgraph query processing and are orthogonal to our approach. For instance, Peregrine [38, 39] uses symmetry breaking (GraphFlow calls this hybrid query plans) to avoid redundant work. However, this is more important in graph mining systems because their workloads have a lot more redundant

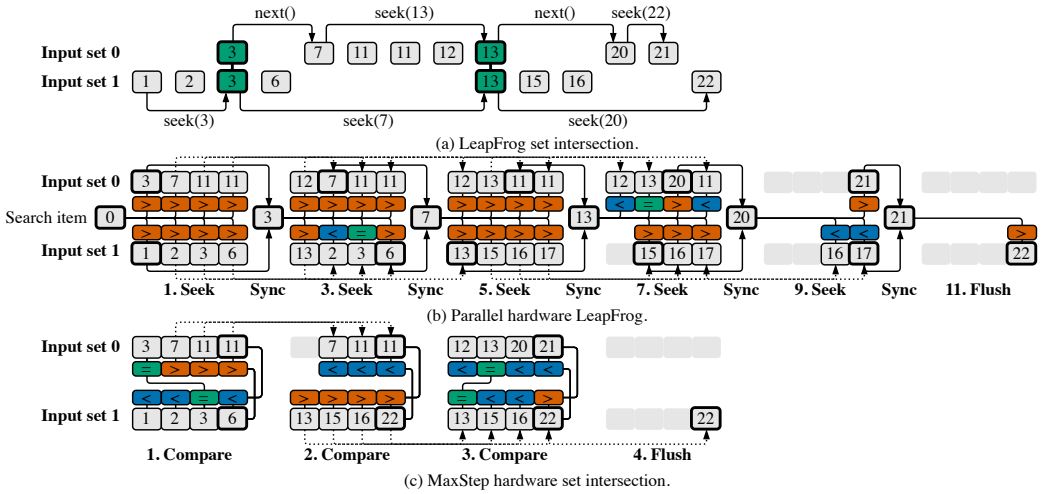


Fig. 3. LeapFrog and MaxStep set intersection approaches.

computations. For subgraph query processing, this only matters if the query graph has symmetries (especially with directed and labeled graphs this is less probable). Additionally, Peregrine tries to optimize isomorphism checks, which in hardware are free, and thus this optimization is not necessary. SISA [14] proposes a set-centric instruction set for processing-in-memory systems but also provides a comprehensive discussion of graph-related accelerators which highlights a gap for graph mining accelerators.

Graph Databases Lastly, there is large body of related work on graph databases covered by surveys [5–7, 13]. Specifically, several approaches have been proposed to enable WCOJs for subgraph query processing in graph databases [8, 37, 41, 72]. However, the focus of these approaches is on adapting the underlying database data structures to enable specialized WCOJ algorithms like Leapfrog Trie Join. To do this, these systems introduce specialized graph indices. We view our work on dedicated subgraph query processing to be orthogonal to the graph database efforts. Our approach can be integrated into a graph database to accelerate these types of queries since our graph data structure and query interface are compatible with other WCOJ implementations.

3 Set Intersections on Steroids

In this section, we design a hardware-accelerated set intersection algorithm. We first describe a hardware adoption of the LeapFrog algorithm before proposing our novel MaxStep hardware set intersection algorithm while comparing the two conceptually. Then, we introduce an optimized implementation of multi-set intersection using MaxStep. Lastly, we show how CPU and GPU implementations of MaxStep and the hardware implementations of LeapFrog and MaxStep compare, and characterize the performance dimensions of the MaxStep hardware set intersection algorithm.

3.1 Set Intersection Approaches

Figure 3 compares LeapFrog (CPU), a parallelized hardware adoption of LeapFrog, and our novel MaxStep hardware set intersection algorithm. The different implementations aim to process k elements of each input set per clock cycle. The parameter k is chosen based on the cache line size (512 bit in our case) divided by the size of the vertex identifier (32 bit unsigned integers in our case)

Algorithm 2 MaxStep set intersection algorithm.

Given: Ordered input sets S, T; Parallelism factor k

```
1:  function INTERSECT
2:      s, t ← 0; R ← {}
3:      while s < |S| or t < |T| do
4:          s_m ← max({S_i, i ∈ [s, s + k)}); t_m ← max(...)
5:          s_n ← s; t_n ← t
6:          parfor i ∈ [0, k)} do
7:              parfor j ∈ [0, k)} do
8:                  if S_{s+i} = T_{t+j} then R ← R ∪ {S_{s+i}}
9:              if S_{s+i} ≤ t_m then s_n ← s_n + 1
10:             if T_{t+i} ≤ s_m then t_n ← t_n + 1
11:         s ← s_n; t ← t_n
12:     return R
```

resulting in a value of 16. A larger value would starve the compute units of data at higher resource utilization while a lower value may make sense in severely resource limited scenarios.

LeapFrog Set Intersection Like all of the algorithms, LeapFrog starts with sorted sets and picks the greatest first element as its search item. With the search item, it loops over the input sets (two in Figure 3(a)). It seeks for the next element that is greater or equal to the search item and picks it as its next search item. If LeapFrog finds an equal element in every input set, it adds this element to the result set and picks the next item as the new search item.

Parallel Hardware LeapFrog We parallelize the comparison operators used in LeapFrog in two steps: seeking for a new search item and syncing the search item between set iterators (cf. Figure 3(b)). In each seek step, the search item is compared against all k elements ($k = 4$ in this example) in each input set. All elements that are less than the search item and the next greatest element are discarded. For the sync step, the next greatest elements of all input sets are compared and the greatest one is used as the next search item. LeapFrog terminates when one set is empty. The remaining elements are flushed. However, parallelized LeapFrog cannot do both steps in one clock cycle in hardware due to timing issues.

Trick. To solve the timing issues caused by too many comparisons at once, we do the seek and sync steps in separate clock cycles. This allows us to put a register between the steps that stores the next search item per set before combining it into the next search item for all sets in the sync step. This removes the comparators and multiplexers used in the sync step from the critical path. Thus, in each iteration (two cycles), parallelized LeapFrog has a guaranteed progress of one element per set.

MaxStep One key observation from parallelizing LeapFrog is that we can implement many more comparisons in parallel in hardware than we can on CPUs. Thus, we propose our novel MaxStep algorithm shown in Algorithm 2 that trades off work optimality (requires more comparisons in total) for improved performance scalability (less synchronization and more parallelism). The notation S_i denotes the i-th element of the set S, max denotes the maximum of the given set, and parfor denotes an unrolled parallel loop. Naively, the algorithm may be implemented by comparing each element of one set against each element of the other set in every clock cycle. However, this would incur k^2 tightly-coupled comparators which cause resource utilization and timing issues in hardware.

Trick. To tackle this issue, we introduce another pipeline stage. In this stage, MaxStep first finds the maximum of the next k elements of the two input sets. This allows us to do all of the complex *less*

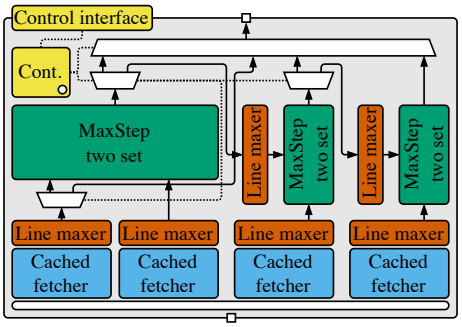


Fig. 4. MaxStep multi-set intersector architecture.

than comparisons (implemented as subtractions in hardware) only against this maximum ($2k \ll k^2$ *less than* operators) removing a multiplexer for k numbers from the critical path. In each iteration, MaxStep then discards all elements of an input set that are less than the maximum value of the other input set. In parallel, it does an equality comparison (much less complex than *less than*) of all elements of input set to all elements of input set and adds them to the result set (k^2 equality operators). Figure 3(c) shows how MaxStep (with $k = 4$) is able to massively reduce the number of iterations (i. e., converge faster) for the same set intersection as parallel hardware LeapFrog. MaxStep guarantees progress of at least k elements of one of the input sets and is thus able to process the same intersection in under half of the clock cycles in this example.

Step-by-step. In step 1 in Figure 3(c), the maxima of the sets are precomputed as 11 and 6. The equality comparison of all combinations of input set elements returns one match for 3 which is passed on to the output set. For input set 1, all elements are smaller than the maximum of input set 0 (i. e., 6). Thus, all elements of input set 1 are discarded and a whole new cache line fetched. For input set 0, only the first element (i. e., 3) is discarded because all other elements are larger than the maximum of input set 1 (i. e., 6). Because there are elements left in the current cache line of input set 0, the maximum for that set stays the same. The new precomputed maximum for input set 1 is 22. We repeat the same computation in step 2 but there are no matches and we can discard the whole input set 0 cache line and keep the input set 1 cache line. We repeat this one last time with a match on 13. We can discard the rest of input set 0 and are left with one element of input set 1 which we flush in the last clock cycle. All *equality* and *less than* comparisons happen in one clock cycle in parallel and the new cache lines arrive on a stream.

3.2 MaxStep Multi-set Intersector Architecture

Figure 4 shows the MaxStep multi-set intersector architecture for four input sets based on the MaxStep operator for two sets. Four fetchers prefetch and pass the input data into the intersector and feed the input lines ($k = 16$ elements per line in the prototype) through line maxers. The line maxers extract the maximum to be used for comparison in the MaxStep operator thus exploiting hardware pipelining. The intersect operators perform the comparisons and pass on the set intersection results. These are fed into a demultiplexer which either forwards the output to the next line maxer and MaxStep operator or to the output port based on a configuration register. As the last component, each MaxStep multi-set intersector contains a controller that is connected to the control interface of the system. The user writes the configuration registers that control the demultiplexers and the multiplexer before the execution. This allows configuration of number of input sets during runtime.

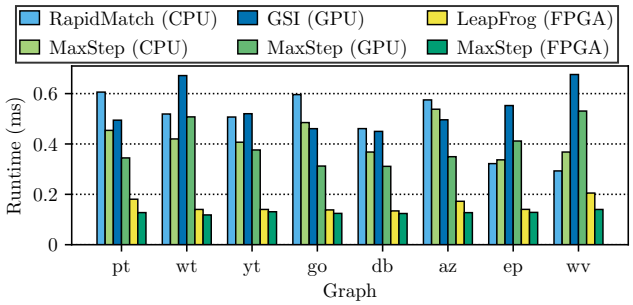


Fig. 5. MaxStep implemented on different processor architectures compared against respective baselines.

The same vertex neighborhoods are often accessed as input sets repeatedly. Thus, we propose a cached fetcher architecture that stores the most recently accessed input set in a cache and serves the subsequent requests to the same input set from the cache. A controller receives the requests and stores the address and size accessed in registers. If the request address and size is equal to the previous one (i. e., same input set; very common access pattern in the GraphMatch architecture), it is flagged as cached and directly inserted into a FIFO queue which multiplexes the output port. If the request cannot be served through the cache, it is forwarded to a buffered fetcher which sends the request to memory, is flagged as fetched, and is inserted into the FIFO queue. When the request is served by memory, the data is cached (implemented as BRAM) as whole lines of memory starting from cache address 0. While serving responses, the data from the buffered fetcher is either directly forwarded or the respective number of memory lines are read from the cache starting at address 0.

3.3 Intersector Performance Characteristics

In the following, we discuss performance characteristics of MaxStep implemented on CPUs, GPUs, and FPGAs in comparison to baselines taken from RapidMatch, GSI, and our LeapFrog hardware implementation. Thereafter, we discuss the performance of MaxStep in detail for the characteristics: input set size, output set size, number of input sets, and ratio of cached requests.

Benefits of Adapting Algorithms to Hardware Figure 5 shows the comparison of the CPU-based RapidMatch set intersection implementation against a vectorized CPU implementation of MaxStep, GSI against a GPU implementation of MaxStep, and our FPGA-based prototype implementations of LeapFrog and MaxStep. The benchmark environment as well as the graphs (cf. Table 3) used for the following experiments are described in Section 5. We ran the GPU measurements on an NVIDIA RTX 3090 using 512 threads. The benchmark shows the runtime in milliseconds for 5000 intersections of neighborhoods of random vertices of the respective graph. The CPU implementation of MaxStep uses the vectorized intersect implementation described in [17] for the equality comparison and works particularly well for very small input sets. However, its performance drops below that of the RapidMatch baseline for the epinions (ep) and wiki-vote (wv) graphs. RapidMatch's hybrid merge-based and galloping [49] set intersection approach works particularly well for these graphs. For the GPU implementations, MaxStep always outperforms GSI because GSI only advances one of the input set pointers per iteration to save a warp summation and their binary search is often slower than a naive linear search on small input sets. LeapFrog (FPGA) and MaxStep (FPGA) perform significantly better than the CPU and GPU implementations. MaxStep (FPGA) always outperforms LeapFrog (FPGA). This is due to the larger progress per clock cycle of MaxStep. In the remainder of this paper, we proceed with the MaxStep multi-set intersector.

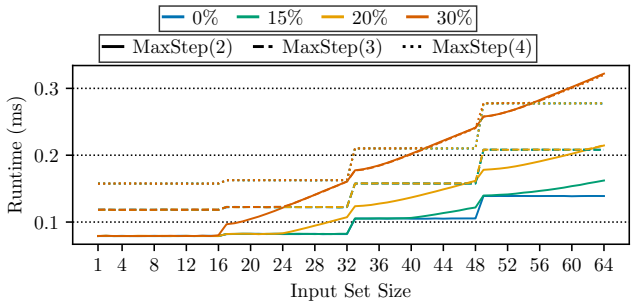


Fig. 6. Runtime of MaxStep for two to four input sets over input set size and output size as percentage of input size.

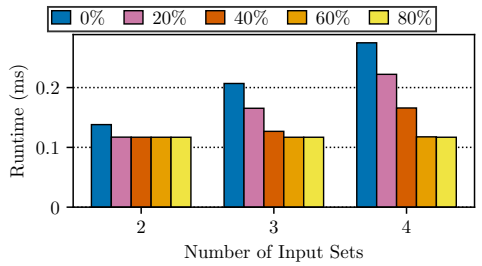


Fig. 7. Input set caching for percentage of accesses cached by number of input sets.

Characteristics of MaxStep Figure 6 shows the runtime of MaxStep in milliseconds with two, three, and four input sets for 5000 set intersections with varying input set and output set sizes. MaxStep on two input sets where 0% of the input set are part of the output set (blue solid line) forms a memory-bound baseline. The baseline is not influenced by fine-granular input set size but by the number of 512bit lines of memory it has to fetch. 16 elements of an input set (32bit values) fit into one line. Thus, the runtime increases each time the last element is part of a new line. For the measurements where 15%, 20%, and 30% of the input sets are in the output sets, the runtime is also bounded by the output set size. This is because MaxStep only puts out one output set element per clock cycle. For three and four input sets, the runtime is higher and increasingly more bound by memory because more data has to be fetched from memory per set intersection. For four input sets, only the measurement where 30% of the input set are part of the output set (green dotted line) is sometimes bound by output size.

Figure 7 shows how input set caching influences the set intersection performance for the MaxStep multi-set intersector for different numbers of input sets. We measure the runtime in milliseconds for 5000 set intersections. Each set intersection works on sets of size 64 without overlap such that the output size does not influence performance. Additionally, we vary the cache hit rate from 0% to 80%. Overall, we observe that with 80% cache hit rate, the performance reaches the same baseline for each number of input sets, thereby denoting a lower bound set by the cycles the hardware needs to perform the intersection itself. For larger numbers of input sets, this baseline is only reached with higher cache hit rate as the intersector is more memory-bound. It is important to note that MaxStep has the same runtime irregardless of the number of input sets if the memory requests do not play a role.

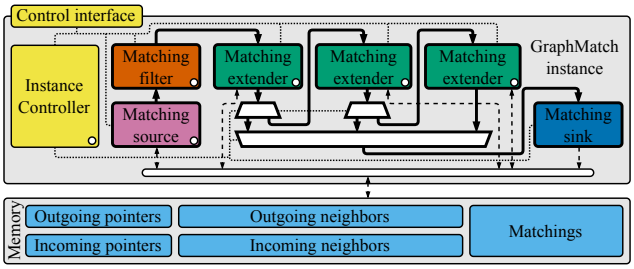


Fig. 8. GraphMatch instance architecture.

3.4 Discussion

We conclude that our MaxStep algorithm is able to leverage parallelism far better than LeapFrog by trading off work optimality for improved performance scaling and is, thus, used in GraphMatch. This is possible due to a trick of splitting off the computation of the maximum of each 512 bit input line into its own pipeline stage which massively reduces the number of complex *less than* comparators used. The performance of MaxStep is bound by memory which we tackle with a cached fetcher that is able to provide better performance for set intersections of repeated input sets.

We also show that the MaxStep algorithm works for CPUs and GPUs. However, for CPUs, a vectorized instruction for equality between two vectors of size k does not exist and we have to resort to a multi-instruction shuffle and compare function which makes MaxStep slower on some graphs. For GPUs, we can perform all comparisons in parallel but have to communicate the results between the threads in a warp with synchronization shuffles. The same step that we are able to perform in hardware in each clock cycle thus takes multiple instructions on CPUs and GPUs.

MaxStep optimizes line 4 of Algorithm 1 and relaxes the worst-case optimality guarantee partially. We preserve the worst-case optimal number of intermediate results. However, MaxStep relaxes the runtime guarantee of the set intersection itself in favor of more parallelism. A binary search-based implementation of set intersection like LeapFrog has a runtime of $O(|X| * log(|Y|/|X|))$ because, in the worst case, for every item in the smaller input set X we need to perform a binary search of size $|Y|/|X|$. With MaxStep, we have a theoretically higher worst-case runtime of $O(|X|/k + |Y|/k)$ where X is again the smaller of the sets and k is the number of elements that fit into one cache line (i. e., 16). However, our approach allows us to do prefetching of sets and extract more parallelism (k). In the real world (cf. Table 3), we mainly see very small neighborhoods as input sets and the parallelism outweighing the higher theoretical runtime.

4 GraphMatch

In this section, we introduce the subgraph query processing architecture of GraphMatch and its components. We then describe how queries can be dynamically switched in GraphMatch during runtime, how we support labeled graphs, and the suitable performance optimizations that we applied to the base system.

4.1 GraphMatch Instance Architecture

Figure 8 shows the architecture of a GraphMatch instance for query graphs with up to five vertices. The flow of partial matchings through the architecture's components is depicted with bold arrows. It starts at the matching source – producing matchings with two query vertices – runs through a matching filter and multiple matching extenders (i. e., a configuration with three extenders results in five vertices overall), demultiplexers and multiplexers (cf. trapezoids) and ends at the matching

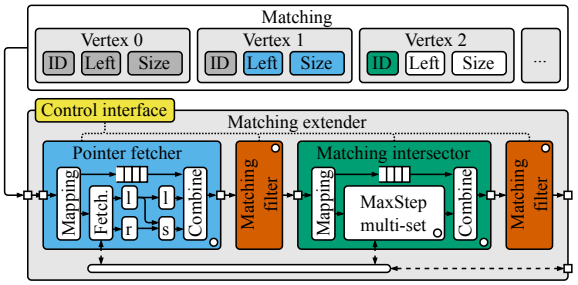


Fig. 9. Matching data type and extender component.

sink. The matching source reads all edges of the graph that form the initial partial matchings. Then the matching filter discards initial matchings that do not fit certain criteria (e. g., vertices not being distinct in the case of graph isomorphisms). Each matching extender then extends the input partial matchings by one query vertex in order of the QVO, which most often means *set intersections*. After each matching extender, there is a matching demultiplexer, which either forwards the incoming partial matchings to the next matching extender, or through a matching multiplexer to the matching sink. Finally, the matching sink writes complete matchings to the designated matchings array in the platforms on-board memory.

In total, the on-board memory contains five data arrays: one CSR data structure consisting of a pointers array and a neighbors array for both incoming and outgoing edges of each vertex and the matchings array. The matching source and matching extenders only read from memory, whereas the matching sink only writes to memory. These accesses are shown in Figure 8 as dashed lines and are combined into one request stream fed to memory by a request merger (cf. white oval box). The request merger also routes the memory read responses back to the corresponding requesters.

The query graph is configured by providing GraphMatch with parameters. All system components with white dots have parameters that are connected to the control interface operated by the user (shown by the dotted lines). The matching source is parameterized with the addresses of the arrays it has to read. The matching filter can be switched on and off, and the matching extenders are each parameterized with the number of neighborhoods and the memory addresses of the neighborhoods they should intersect. Finally, the instance controller manages the query's execution. It has a parameter for query size that switches the demultiplexers and multiplexer, is responsible to trigger the execution when all components are ready, and finally returns statistics like runtime to the CPU.

Matching Data Type Figure 9 depicts the component architecture of a matching extender in the context of the matching data type. Matchings consist of a configurable number of vertices with a vertex identifier, left bound for pointers, and size that denotes the neighborhood size. The number of vertices in the matching data type is equal to the number of vertices in the GraphMatch instance (e. g., five for the instance in Figure 8). The left bound and neighborhood size are kept in the matching as metadata between matching extenders. Only when the query first requires intersection on the outbound edges of a vertex and thereafter an intersection on the inbound edges or the other way around, we have to fetch new metadata from the respective pointers array. The metadata is discarded when writing to memory in the matching sink, since only the vertex identifiers are relevant for the result.

Matching Extender The matching extender component at level l takes a matching with l vertices where the vertex at position l is still missing the metadata. It then fetches the required information and tries to extend the matching to vertex $l + 1$. To do this, the partial matching is acquired through

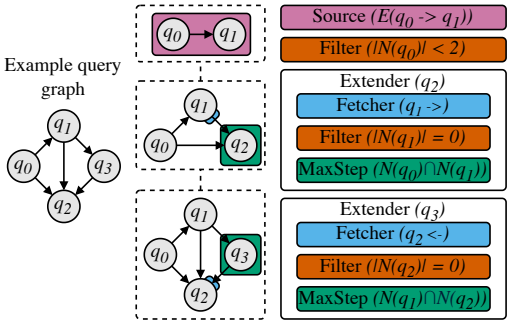


Fig. 10. GraphMatch query parameters for query Q5.

a pointer fetcher that retrieves the required metadata, a first matching filter, a matching intersector performing the set intersection required and extending the matching by a vertex, and lastly another matching filter. The pointer fetcher takes a partial matching and a mapping as a parameter and fetches the respective metadata. Dictated by the mapping, a buffered fetcher (Fetch.) fetches the lines containing the pointers at positions v and $v + 1$, where v is the vertex identifier. These form the left (l) and right (r) bound of the neighborhood. The pointer fetcher subtracts l from r to get the neighborhood size and, finally, combines the new metadata with the partial matching. The first matching filter filters out empty sets, i. e., sets where any of the neighborhood sizes used in the following intersection are 0. The matching intersector maps the matching vertices to intersector spots in the MaxStep intersector, specified in Section 3, based on mapping parameter extracted from the query graph. For example, if there is an intersection between vertices 0 and 2, the MaxStep intersector is configured to do a 2 set intersection mapping vertex 0 to spot 0 and vertex 2 to spot 1. During the intersection, the partial matching is stored in a FIFO and the combined subcomponent extends the current partial matching until the intersection is finished. It then proceeds with the next partial matching from the FIFO. Depending on whether the workload is a graph isomorphism or homomorphism, the second matching filter sieves out matchings for which the newly added vertex is not unique in the matching.

4.2 Dynamic Queries & Labeled Graphs

Based on the GraphMatch architecture, we specify dynamic queries with our query parser, and transformations of a query graph as query parameters for GraphMatch. This also allows us to support labeled graphs by partitioning the data graph.

Figure 10 shows how the example query on the left side is deconstructed by the query parser to get the GraphMatch parameters to map the query graph. The instance controller receives the number of query vertices and the address of the matchings array. Each query starts with two vertices connected via an edge. These form the parameters that are mapped to the matching source. The parameters are the addresses of the outgoing pointers and neighbors arrays of q_0. Additionally, the matching filter after the matching source receives the neighborhood size of q_0 in the complete query graph as a parameter. In this example, the neighborhood has to be at least size two. The first matching extender is parameterized with a mapping to fetch the metadata for q_1 and the address to outgoing pointers for the pointer fetcher. Furthermore, the matching intersector receives a mapping to intersect the neighborhoods of q_0 and q_1 as a two-set intersection on outgoing neighbors. Finally, in the second matching extender, the pointer fetcher should load the incoming pointers for q_2 and we pass a mapping parameter to intersect the neighborhoods of q_1 and q_2 as a two-set intersection

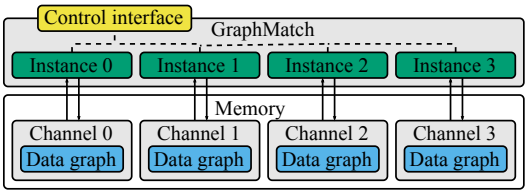


Fig. 11. GraphMatch multi-instance scaling.

on outgoing neighbors. If a query is larger than the maximum number of query vertices of the instance, we can materialize the partial matchings into memory, read them back to the beginning of the matching extender pipeline and feed them through the levels again. However, details of this are beyond the scope of this work.

To support labeled graphs, we partition the data graph into one subgraph per vertex label pair such that each of these subgraphs only contains edges from the source vertex label to the destination vertex label of the pair. Similarly, we are able to support edge labels by partitioning the graph into partitions where edges match the given edge label. By passing the corresponding addresses to the pointer fetcher and matching intersector based on the vertex labels of the query vertices, we can then map to labeled subgraph graph queries. To optimize the size of the subgraphs, we sort the vertices by vertex label as a preprocessing step. Thus, all vertices of a certain label are a range in the set of vertices and for every subgraph we only need to store this range of vertices. To make the memory accesses line up we have to subtract the identifier of the first vertex in a range of labels from the address such that we can look up pointers for vertices than can only be in that range.

4.3 Optimizations

In the following, we explain the four key optimizations for GraphMatch: input set caching, failing set pruning, instance parallelization, and stride mapping.

Input Set Caching As a first optimization, we implement caching (cf. Section 3.3). We suspect that locality of reference exists in the input set of the MaxStep intersector, as found in each matching intersector, as well as in the input of the pointer fetcher. This is especially the case when new metadata for existing vertices has to be loaded for a partial matching. Thus, all instances of the MaxStep intersector and the pointer fetcher employ caching.

Failing Set Pruning As a second optimization, we introduce failing set pruning. In GraphMatch, we implement it inside the matching filter, after the matching source, and in the first matching filter of each matching extender. In addition to filtering out empty sets, we can also filter partial matchings when the neighborhood of a vertex is not at least as big as the corresponding vertices neighborhood in the query graph. For example, for graph isomorphisms on Q5 (cf. Figure 13), for q_0 the neighborhood size needs to be at least 2. Thus, we parameterize the matching filters, such that we can customize them for the query vertex neighborhood size.

Instance Parallelization GraphMatch may be scaled to multiple instances each assigned its own memory channel and otherwise only connected to the mutual control interface. The data graph is copied to each memory channel such that the instances can work independent. The system can, thus, either process one query on one data graph in parallel such that the vertex set is split up into four intervals each assigned to one of the four instances, or process different queries and data graphs on the four instances independently (especially interesting in multi-tenant environments). Since instances run on separate memory channels, there is no resource contention and perfect performance isolation during the execution.

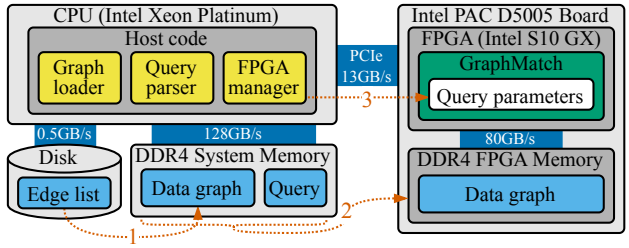


Fig. 12. FPGA prototype architecture.

Stride Mapping Lastly, we apply stride mapping [25] to improve load balancing across GraphMatch instances. As the GraphMatch instances cannot communicate partial matching among each other, each vertex interval should require roughly equal amount of work. However, this is not a realistic assumption when working with real-world graphs [26], which are often skewed. With stride mapping, we do a light-weight vertex reordering. Stride mapping is a technique for semi-random shuffling of the vertex identifiers before partitioning to create a new vertex ordering with a constant stride. In our case we use a stride of 100, which results in a new vertex order $v_0, v_{100}, v_{200}, ...$, which virtually results in each vertex being mapped to a different hardware instance.

4.4 Performance Model

The effectiveness of input set caching showed that memory accesses are the bottleneck of GraphMatch. Thus, we propose a performance model that is based on the number of memory requests with l denoting the number of vertex identifiers or pointers that fit into the width of the memory interface. For instance, for 32bit values and a 512bit wide memory interface, l equals 16. To materialize the initial edges, we need the following number of memory requests:

$$M = (|V| + 1)/l + |E|/l \quad . \tag{1}$$

We need to read $|V| + 1$ pointers and $|E|$ edges sequentially. Thus, we can divide these numbers by l. For each extension of this partial matching we need approximately this additional amount of memory requests, where f is the number of vertices for which new pointers have to be fetched, m is the number of partial matchings going into this extension, and s is the number of sets being intersected:

$$M = f \cdot m + s \cdot (m \cdot D_{avg}/\max(l, D_{avg})) \quad . \tag{2}$$

We need to fetch $f \cdot m$ pairs of pointers which most often are part of one line of memory such that we only need one request. For each intersector, we need to make $m \cdot D_{avg}/\min(l, D_{avg})$ requests. The max term is there in case the neighborhood is smaller than l where we still have to fetch a whole line. However, this may be lowered with caching. For example, $s = 1$ for an extension by one edge, and $m = |E|$ for the first extension after the matching source.

5 Evaluation

We start this section by introducing the prototype architecture and the experiment setup. We then compare GraphMatch against (i) the state-of-the-art CPU-based systems GraphFlow and RapidMatch on unlabeled graphs, and (ii) GraphFlow, RapidMatch, and the fastest accelerator systems (FAST, GSI, and GpSM) on labeled graphs. We also report results for scaling GraphMatch from 1 up to 4 instances, before evaluating the effects of the four optimizations.

Table 2. FPGA resource utilization and clock frequency for three prototype configurations.

System	p	l	c	LUTs	Regs.	BRAM	DSPs	Clock freq.
GraphMatch	4	6	☐	48%	26%	21%	0%	187MHz
	4	6	☑	54%	33%	34%	0%	191MHz
	4	7	☑	70%	43%	45%	0%	167MHz

p: Instances; l: Maximum query size; c: Input set caching; LUTs: Look-up tables; Regs.: Registers; BRAM: Block RAM; DSPs.: Digital signal processors; Clock freq.: Clock frequency; ☑: yes; ☐: no

Prototype Architecture Figure 12 shows how the GraphMatch prototype is deployed for the benchmarks. The prototype is implemented on a system that features an FPGA running GraphMatch and FPGA memory used as intermediate data storage for the data graph during subgraph query processing. A CPU loads and prepares the data graph, parses and programs the query graph to GraphMatch, and manages the execution on the FPGA. The CPU could be replaced with a client directly communicating with GraphMatch over the network, completely removing the CPU from the system, since GraphMatch can handle arbitrary queries during runtime.

For our experiments, we work with a server equipped with an Intel Programmable Accelerator Card (PAC) D5005 attached via PCIe v3. The system features two Intel Xeon Gold 6142 CPUs at 2.6GHz and 384GB of DDR4-2666 memory, while the D5005 board is equipped with four channels of DDR4-2400 memory with a total capacity of 32GB and a bandwidth of 76.8GB/s. The design itself is integrated into Intel Open Programmable Execution Engine (OPAE) and is synthesized with Quartus 19.4.

Hardware Configurations We use three different GraphMatch prototype configurations for the benchmarks. Table 2 shows the three different GraphMatch prototype configurations used for the benchmarks. One variant is without input set caching (c) and two with input set caching for the pointer fetchers and MaxStep set intersectors. All variants have $p = 4$ instances of GraphMatch, one for each memory channel, with a maximum query graph size of $l = 6$ and $l = 7$ for the labeled graph benchmarks. Note that, without further resource utilization optimization, up to 8 instances could be placed onto the chip. All types including pointers, vertex identifiers, and labels are 32bit unsigned integers. The Quartus power estimator estimates a thermal design power (TDP) of 61.7W for GraphMatch with four instances. This is significantly lower than the Intel CPU's 150W and the 350W TDP of the RTX 3090 used for the GPU benchmarks in Section 3.3.

Data and Query Graphs Table 3 show the properties of the unlabeled and labeled graph data sets used to benchmark GraphMatch (including the storage size of the graph on disk and the time to transfer the graph to the FPGA). This selection represents the most important graphs considered by the state-of-the-art subgraph query processing systems (cf. Section 2.3). We additionally show graph properties like size ($|V|$ and $|E|$), average degree (D_{avg}), ratio of vertices in the largest strongly-connected component (SCC), and number of vertex labels ($|L|$) that are useful to explain different performance effects observed in the benchmarks. For the labeled graph benchmark, we generated four scale factors (1, 3, 10, and 60) of labeled graphs also used in [42] with the LDBC social network benchmark (SNB) graph generator version 0.3.2 [65]. We converted the generated graphs into an edge list with labeled vertices and made the person knows person edges undirected for semantical consistency. We note that to convert the snb_{60} graph to the GraphFlow format we needed to manually set the Java heap space available to GraphFlow to 256GB which is an order of magnitude more than the same graph stored on disk. For the intersection benchmark, we generated different configurations of a synthetic graph $\text{syn}_{n,d}$ with another parameter for intersection size between

Table 3. Graphs used often by systems in Table 1 (real-world graphs from [50] and [61]).

Name	$\|V\|$	$\|E\|$	D_{avg}	ø	SCC	$\|L\|$	File size	Load
patents (pt)	3.8M	16.5M	4.34	22	1.00	1	281MB	990ms
wiki-talk (wt)	2.4M	5.0M	2.10	11	0.05	1	66.5MB	829ms
youtube (yt)	1.2M	3.0M	5.16	20	0.98	1	38.7MB	822ms
google (go)	875.7K	5.1M	5.82	21	0.50	1	75.4MB	829ms
dblp (db)	426.0K	1.0M	4.93	21	0.74	1	13.9MB	409ms
amazon (az)	403.3K	3.4M	8.43	21	0.98	1	47.9MB	415ms
epinions (ep)	75.9K	508.8K	6.70	14	0.43	1	5.66MB	2.70ms
wiki-vote (wv)	7,115	103.7K	14.56	7	0.18	1	1.10MB	1.75ms
snb_1	3.2M	17.4M	5.44	-	-	11	200MB	2.65s
snb_3	9.3M	52.7M	5.67	-	-	11	640MB	2.79s
snb_{10}	30.0M	176.6M	5.89	-	-	11	2.35GB	2.98s
snb_{60}	187.1M	1.25B	6.68	-	-	11	18.3GB	3.99s
$syn_{n,d}$	n	$n \cdot d$	d	-	1	1	-	-

SCC: Ratio of vertices in largest strongly-connected component; Load: Time to load the graph into FPGA memory

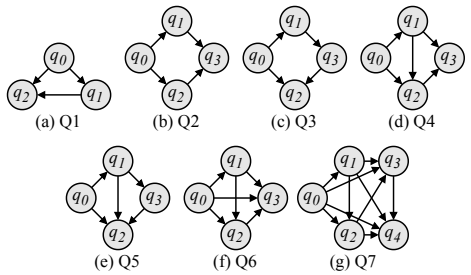


Fig. 13. Unlabeled query graphs (adopted from [53]).

two adjacent vertices. When loading the data graphs, we transform the set of vertex identifiers to be dense (i. e., excluding vertices that have degree 0).

Figure 13 shows the unlabeled queries we use in our evaluation comparing against GraphFlow and RapidMatch, taken from [53]. These can be classified as cliques (Q1, Q6, and Q7), cycles (Q1, Q2, and Q3), and other graphs (Q4 and Q5). Figure 14 shows the labeled query graphs that were adopted from the LDBC SNB benchmark version 0.3.2 [4] in [46] and used in [42]. It was not clear from their descriptions how the queries were constructed because they were described as undirected query graph but there are edges between tag classes (tCls) and persons (psn) and posts (post) (e.g., SQ2 and SQ5) that do not have the same semantics when undirected. Thus, we added the edge labels and directions that we used for our experiments. Furthermore, the edges between persons are the knows relation which we made undirected for semantical consistency.

5.1 Comparison to GraphFlow & RapidMatch

The state-of-the-art systems are GraphFlow and RapidMatch (cf. Section 2), which we use as baselines for comparison.

GraphFlow Figure 15 compares the performance of four instances of GraphMatch to GraphFlow [53] running on 16 threads (a full CPU on our server) with `numactrl` restricting it to one node. The plots show the performance in seconds on a logarithmic scale. For GraphMatch, we picked the

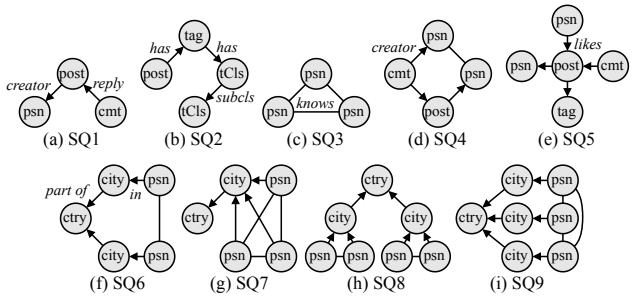


Fig. 14. Labeled query graphs (adopted from [42]).

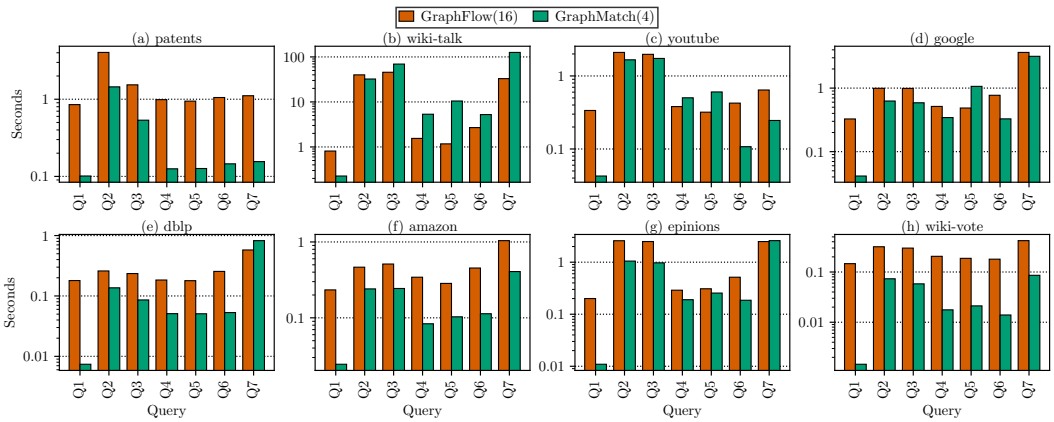


Fig. 15. GraphMatch (4 instances) vs. GraphFlow (16 threads) on directed, unlabeled graphs for subgraph homomorphisms.

best QVOs for each query and data graph combination. GraphFlow uses directed graphs, however, only supports subgraph homomorphisms. Thus, to set the ground for a fair comparison, we have turned off distinct vertex checking and changed the failing set pruning optimizations to match the workload in GraphMatch. We note that for GraphFlow, we had to execute three scripts that prepare the graphs beforehand for each graph. It was not clear to us how much these scripts optimize the graph layout. When looking at the performance numbers, overall, we observe big performance improvements with GraphMatch. This is especially pronounced for Q1 on all data graphs. GraphMatch also performs exceptionally well for Q4-Q6 on many graphs. Another interesting observation is that Q2 and Q3 pose a challenge for GraphMatch on all graphs. We attribute that to their low intermediate selectivity, which leads to a large number of partial matchings after extending to q_2. Finally, we note that the graph structures of wiki-talk and youtube are more problematic for GraphMatch than for GraphFlow. These graphs exhibit highly exponential (i. e., skewed) degree distributions, which lead to some very large vertex neighborhoods that are used often in partial matchings. These cannot be cached with our design due to its relatively small caches, but could be cached by the more sophisticated cache hierarchy of CPUs. In particular, GraphMatch outperforms GraphFlow by an average of 2.68× for all graphs with a maximum of 100× for query-1 on wiki-vote.

RapidMatch Figure 16 compares four instances of GraphMatch to RapidMatch [67]. We were not able to find any configuration parameter to configure the number of threads RapidMatch

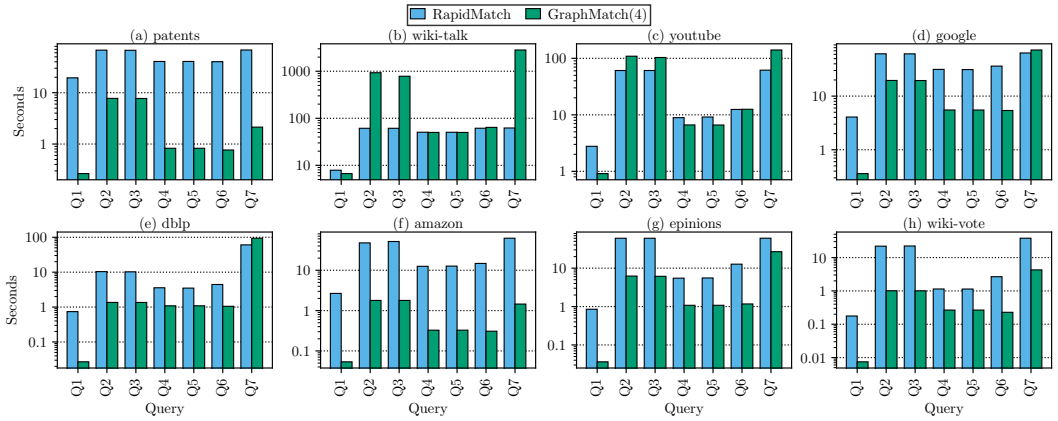


Fig. 16. GraphMatch (4 instances) vs. RapidMatch on undirected, unlabeled graphs for subgraph isomorphisms.

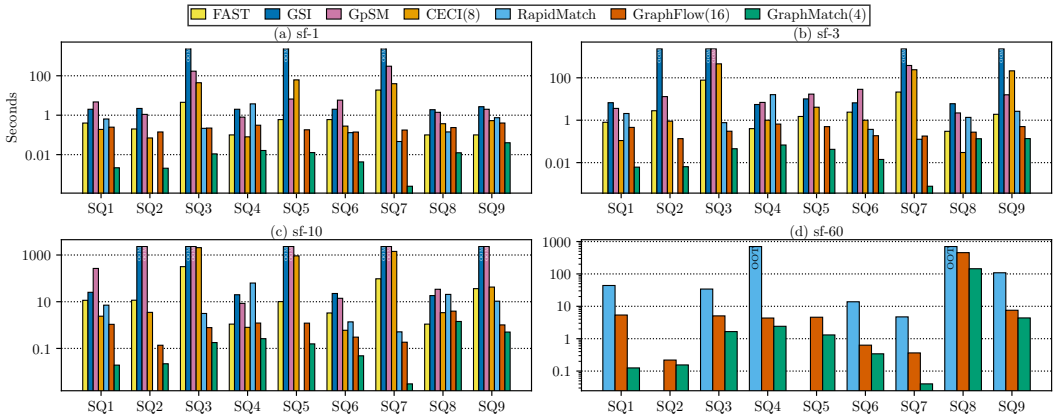


Fig. 17. CPU, GPU, and FPGA systems comparison on directed, labeled graphs for subgraph isomorphisms.

runs on, so it only uses vectorized instructions for intra-set intersection parallelism. RapidMatch only works with undirected graphs, so we also make the graphs undirected for GraphMatch and compute sub-graph isomorphisms (even though RapidMatch can also compute homomorphisms). Overall, we observe significantly better performance for GraphMatch compared to RapidMatch, with an average speedup of 5.16×. Similar to the comparison to GraphFlow, GraphMatch performs exceptionally well for Q1. As before, the system performs less well for Q2, Q3, and Q7 because the CPU can cache neighborhoods more effectively whereas GraphMatch cannot share cached neighborhoods between matching extenders. We further note that the performance results for Q2 and Q3, as well as for Q4 and Q5 are similar, because in an undirected graph the orientation of the query edges makes no difference. Once again, we note the wiki-talk and youtube data graphs are challenging for GraphMatch because of their heavily exponential degree distribution. This causes the cached fetchers to run out of space for large neighborhoods that are used often.

5.2 LDBC SNB Benchmark Comparison

Figure 17 compares seven subgraph query processing systems (cf. Table 1) on the four LDBC SNB graphs. The measurements for FAST, GSI, GpSM, and CECI are taken from the FAST paper [42]. However, even though the FAST paper does experiments on the snb_{60} graph, we were not able to obtain the raw numbers to plot them. Thus, we only show RapidMatch, GraphFlow, and GraphMatch results for the snb_{60} graph. For RapidMatch, we were not able to obtain measurements for SQ2 and SQ5 because the system does not support directed graphs. Additionally, we set a time limit of 10 minutes for RapidMatch and the system ran out of time (OOT) for queries SQ4 and SQ8 on the snb_{60} graph. Overall, GraphMatch is the fastest system in all measurements except on snb_3 query SQ8 and snb_{10} query SQ8. The SQ8 query allows for query plan optimizations that GraphMatch does not support in the current prototype but we view as an orthogonal optimization. On average, GraphMatch outperforms the fastest other system GraphFlow by a factor of 15.84×. The GPU systems perform the worst due to algorithmic and data structure inefficiencies. GpSM performs two joins for each query vertex instead of just one to be able to exploit the GPU parallelism. GSI overcomes this limitation, but introduces a hashing indirection that induces additional memory accesses. Furthermore, both GPU systems write intermediate results to memory which can become magnitudes larger than the result size making them run out of memory (OOM) even on relatively small graphs. GraphMatch solves this problem with its data flow architecture that heavily optimizes the issued reads and writes to memory. Additionally, GraphMatch achieves an average 112× speedup over the CPU-FPGA system FAST. This is due to both algorithmic and architectural advantages that we explain on the example of SQ5 in Figure 17(c). For most queries, the WCOJ-based approach vastly outperforms the backtracking-based approach because it reduces the number of intermediate results as much as possible. Thus, GraphFlow outperforms CECI by a factor of 835.5× while FAST can only achieve a speedup of 91× over CECI with the same backtracking-based approach on a CPU-FPGA system. GraphMatch combines the superior WCOJ-based approach with our novel MaxStep algorithm and a data flow architecture for a speedup of 7.0× over GraphFlow to achieve an overall performance improvement of 64.5× over FAST, in this example. Additionally, FAST is a hybrid CPU-FPGA system and thus requires costly communication between CPU and FPGA. The improvements are a combination of many things and may be influenced by hardware selection. However, while this is not a perfect apples-to-apples comparison, the Xilinx Alveo U200 card that FAST uses was introduced only a year prior to the card that we use for our experiments and is comparable technology-wise.

5.3 GraphMatch Scalability & Optimizations

Figure 18 shows the scalability of GraphMatch as we scale from one up to four instances. We report the speedup over a baseline using a single instance for all data graphs and queries. When using a single instance, the stride mapping optimization is disabled because it makes no difference for measurements on a single instance. Otherwise, all optimizations discussed in Section 4.3 are enabled by default. The measurements show that the speedup is mostly dependent on the properties of the data set itself. For example, we observe a linear scalability on the patents and amazon graphs, which is not the case for the other graphs where scalability is influenced by the number of intermediate (and actual matchings) in the range of vertices assigned to an instance. This effect is particularly pronounced for the wiki-talk and youtube graphs. Scalability could be improved in future work, e. g., with work-stealing where an idling instance takes over partial matchings from a busy instance to balance out the load. However, this would also introduce dependencies between the instances and could lead to higher design complexity.

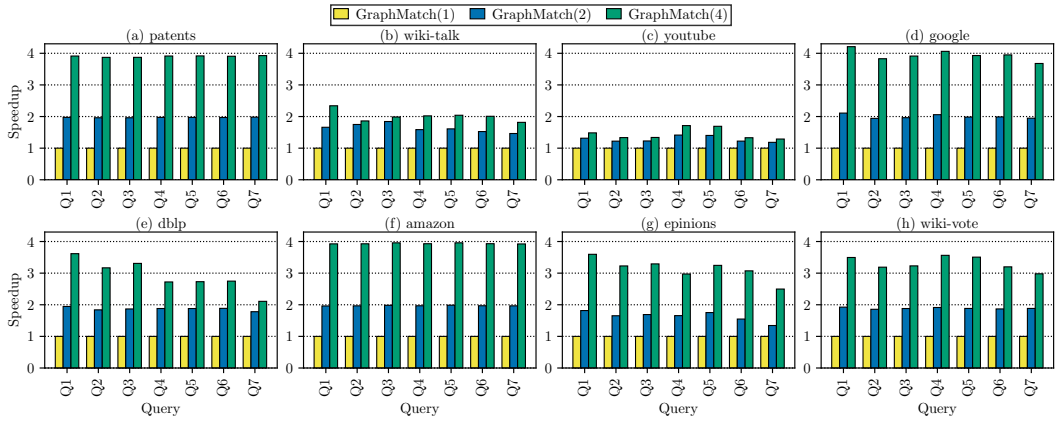


Fig. 18. Scalability of GraphMatch from 1 to 4 instances.

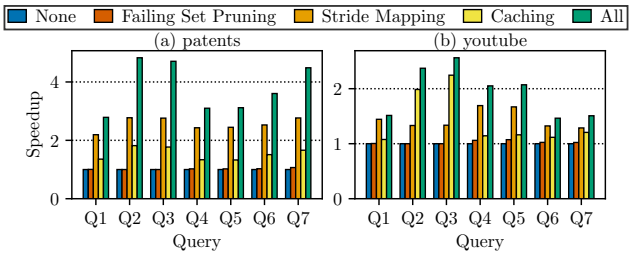


Fig. 19. Effects of GraphMatch optimizations.

Figure 19 shows the effects of failing set pruning, stride mapping, and input set caching on the example of the patents and the youtube query graphs. The baseline is GraphMatch with four instances with all optimizations turned off (None). The failing set pruning optimization effectiveness depends on the query graph. For most query graphs it has a small effect but gets more effective with larger query graphs (e. g., Q7). Stride mapping has the largest effect because it balances out the work required between the GraphMatch instances. The effectiveness of input set caching mostly depends on the data graph. It has the biggest effect on the youtube data graph benchmarks. The optimizations work well together to provide a significant performance boost when all of them are turned on (All).

5.4 Limitations on Query & Data Graph Size

Our prototype limits the size of the query graph to the depth of the GraphMatch pipeline and set intersections to a maximum of four sets. However, the pipeline depth is configurable during synthesis and the query size is configurable during runtime within the boundaries of the synthesized design. While the resources available on the chip limit the pipeline depth, this will be alleviated by newer hardware (e. g., AMD V80 [3]). For queries larger than the number of pipeline steps, the query may also be partitioned into runs through the pipeline. After each run, the partial results need to be materialized into memory and streamed back into the pipeline for the next partition of the query graph. The design currently only lacks the module for the replay of partial matchings back into the pipeline to support this.

On the system available to us, the size of the data graph is limited by the available 64GB of FPGA memory. This limitation can be subverted with three approaches: (i) larger memory attached to the FPGA, (ii) cache-coherent interconnects that allow access to CPU memory, and (iii) shared memory approaches on the FPGA. An already existing platform that features 512GB of FPGA memory and a cache-coherent interconnect is Enzian [24]. Shared memory is also available through fault based page migration [45] and shells that implement this like Coyote [59]. We also expect FPGA hardware with compute express link (CXL) in the near future.

5.5 Discussion

GraphMatch exhibits linear scalability when the graphs have close to uniform degree distribution. Overall, we show improvements over state-of-the-art systems GraphFlow and RapidMatch with an average (unlabeled and labeled graphs) speedup of 6.98× and 17.08× respectively. Additionally, the performance can be significantly improved with our proposed optimizations (up to 5× and 3× on the patents and youtube dataset, respectively). When looking at the performance of individual queries, we observe that GraphMatch exhibits excellent performance for graphs that have only slightly skewed degree distributions but performance drops for queries Q2 and Q3 on highly skewed graphs (e.g., youtube and wiki-talk).

Even though FPGAs can usually only be utilized up to ~80%, the resource utilization of GraphMatch still leaves room for additional functionality. For example, integrating GraphMatch with a network stack [60] or driver for SSDs to load the data graph without CPU involvement could make GraphMatch independent of a host CPU. If the clock frequency of these other components is not lower than that of GraphMatch, there would be no direct performance disadvantage. However, adding another heavy component like a graph traversal engine [26] working on the same graphs to the design would be difficult due to contention on the memory bandwidth. For resource constrained use cases, smaller instances could trade off performance for less resource utilization.

Future work could improve performance with sophisticated caching (e. g., a second caching layer with more capacity that would also be able to cache input sets across matching extenders). Porting the design to a system with HBM could significantly improve performance but comes with its own resource utilization challenges [28]. Lastly, we consider connecting the GraphMatch instances with a work-stealing enabled matching crossbar to tackle workload imbalances and exploring query plan optimization (e. g., hybrid binary join/WCOJ queries).

6 Conclusion

We propose GraphMatch, a hardware-accelerated subgraph query processing system based on WCOJ prototyped on FPGAs. GraphMatch is inspired by the insight that current CPU-based subgraph query processing systems are limited by set intersection. We first introduce the novel set intersection algorithm MaxStep that is able to outperform the state-of-the-art CPU intersectors. Thereafter, we introduce the GraphMatch subgraph query processing architecture with dynamic query reconfiguration and four key performance optimizations, outperforming current state-of-the-art systems.

In future work, we will extend GraphMatch's capabilities with multi-hop edges, non-edges, and property graphs.

Acknowledgments

We thank Intel® Corp. and Lenovo™ for the hardware provided to conduct this project.

References

[1] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79.

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44.

[3] AMD. 2024. AMD Alveo V80 Compute Accelerator Card. https://www.amd.com/content/dam/amd/en/documents/products/accelerators/alveo/v80/alveo-v80-product-brief.pdf.

[4] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.

[6] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39.

[7] Renzo Angles and Claudio Gutierrez. 2018. An Introduction to Graph Data Management. In *Graph Data Management, Fundamental Issues and Recent Developments*. Springer, 1–32.

[8] Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2024. The Ring: Worst-case Optimal Joins in Graph Databases using (Almost) No Extra Space. *ACM Trans. Database Syst.* 49, 2 (2024), 5:1–5:45.

[9] Amazon AWS. 2024. Amazon Neptune. https://aws.amazon.com/neptune/. accessed October 14, 2024.

[10] Amazon AWS. 2024. Microsoft Cosmos DB for Apache Gremlin. https://learn.microsoft.com/en-us/azure/cosmos-db/gremlin/introduction. accessed October 14, 2024.

[11] Jeff Barr. 2021. AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries. https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/. accessed October 14, 2024.

[12] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. In *AAAI*. 17682–17690.

[13] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2024. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* 56, 2 (2024), 31:1–31:40.

[14] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez-Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO*. ACM, 282–297.

[15] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*. ACM, 1447–1462.

[16] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*. ACM, 1199–1214.

[17] Guille D. Canas. 2021. Faster-Than-Native Alternatives for x86 VP2INTERSECT Instructions. *CoRR* abs/2112.06342 (2021).

[18] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (2023), 441–454.

[19] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 7:1–7:13.

[20] Joanna Che, Kasra Jamshidi, and Keval Vora. 2024. Contigra: Graph Mining with Containment Constraints. In *EuroSys*. ACM, 50–65.

[21] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *ASPLOS*. ACM, 413–426.

[22] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *ICS*. ACM, 378–391.
[23] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *PVLDB* 13, 8 (2020), 1190–1205.
[24] David A. Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: an open, general, CPU/FPGA platform for systems software research. In *ASPLOS*. ACM, 434–451.
[25] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *FPGA*. 217–226.
[26] Jonas Dann, Daniel Ritter, and Holger Fröning. 2022. GraphScale: Scalable Bandwidth-Efficient Graph Processing on FPGAs. In *FPL*. IEEE, 24–32.
[27] Jonas Dann, Daniel Ritter, and Holger Fröning. 2023. Non-relational Databases on FPGAs: Survey, Design Decisions, Challenges. *ACM Comput. Surv.* 55, 11 (2023), 225:1–225:37.
[28] Jonas Dann, Daniel Ritter, and Holger Fröning. 2024. GraphScale: Scalable Processing on FPGAs for HBM and Large Graphs. *ACM Trans. Reconfigurable Technol. Syst.* 17, 2 (2024), 22:1–22:23.
[29] Jonas Dann, Royden Wagner, Daniel Ritter, Christian Faerber, and Holger Fröning. 2022. PipeJSON: Parsing JSON at Line Speed on FPGAs. In *DaMoN*. 3:1–3:7.
[30] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*. ACM, 365–376.
[31] Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2024. Talk like a Graph: Encoding Graphs for Large Language Models. In *ICLR*. OpenReview.net.
[32] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB* 13, 11 (2020), 1891–1904.
[33] Per Fuchs, Peter A. Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *GRADES-NDA*. ACM, 4:1–4:11.
[34] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. 2021. SumPA: Efficient Pattern-Centric Graph Mining with Pattern Abstraction. In *PACT*. IEEE, 318–330.
[35] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*. ACM, 1429–1446.
[36] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *SIGMOD*. ACM, 1587–1602.
[37] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11778)*. Springer, 258–275.
[38] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *EuroSys*. ACM, 13:1–13:16.
[39] Kasra Jamshidi and Keval Vora. 2021. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 1–10.
[40] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. 2024. Accelerating Graph-based Vector Search via Delayed-Synchronization Traversal. *CoRR* abs/2406.12385 (2024).
[41] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *CIDR*. www.cidrdb.org.
[42] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, and You Peng. 2021. FAST: FPGA-based Subgraph Matching on Massive Graphs. In *ICDE*. IEEE, 1452–1463.
[43] Marko Kabic, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3, 3 (2025), 187:1–187:25.
[44] Marko Kabić, Bowen Wu, Jonas Dann, and Gustavo Alonso. 2025. Powerful GPUs or Fast Interconnects: Analyzing Relational Workloads on Modern GPUs. *Proc. VLDB Endow.* 18, 11 (2025), 4350–4363. doi:10.14778/3749646.3749698
[45] Torben Kalkhof and Andreas Koch. 2021. Efficient Physical Page Migrations in Shared Virtual Memory Reconfigurable Computing Systems. In *FPT*. IEEE, 1–10.
[46] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *PVLDB* 12, 10 (2019), 1099–1112.
[47] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. 2022. Bandwidth-optimal Relational Joins on FPGAs. In *EDBT*. 1:27–1:39.

[48] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *PVLDB* 6, 2 (2012), 133–144.
[49] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exp.* 46, 6 (2016), 723–749.
[50] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[51] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 21–37.
[52] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Wolfgang Lehner. 2025. SAP HANA Cloud: Data Management for Modern Enterprise Applications. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, to appear.
[53] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB* 12, 11 (2019), 1692–1704.
[54] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2023. BOSS - An Architecture for Database Kernel Composition. *Proc. VLDB Endow.* 17, 4 (2023), 877–890.
[55] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40.
[56] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16.
[57] Natasa Przulj, Derek G. Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinform.* 22, 8 (2006), 974–980.
[58] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2016. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM* 59, 11 (2016), 114–122.
[59] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. 2025. Coyote v2: Raising the Level of Abstraction for Data Center FPGAs. *CoRR* abs/2504.21538 (2025).
[60] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. 2025. RoCE BALBOA: Service-enhanced Data Center RDMA for SmartNICs. *CoRR* abs/2507.20412 (2025).
[61] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. http://networkrepository.com. In *AAAI*.
[62] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.
[63] SAP SE. 2021. SAP HANA Graph Reference. https://help.sap.com/doc/21574acf46fe45a8ae9def213f2c4d9e/2.0.06/en-US/SAP_HANA_Graph_Reference_en.pdf. accessed October 14, 2024.
[64] Tom AB Snijders, Philippa E Pattison, Garry L Robins, and Mark S Handcock. 2006. New specifications for exponential random graph models. *Sociological methodology* 36, 1 (2006), 99–153.
[65] Ben Steer. 2020. LDBC Social Network Benchmark Data Generator. https://hub.docker.com/r/ldbc/datagen. accessed October 14, 2024.
[66] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *SIGMOD*. ACM, 1083–1098.
[67] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *PVLDB* 14, 2 (2020), 176–188.
[68] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *SOSP*. ACM, 425–440.
[69] Ha Nguyen Tran, Jung-Jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *DASFAA (Lecture Notes in Computer Science, Vol. 9049)*. Springer, 299–315.
[70] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.
[71] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR* abs/1210.0481 (2012).
[72] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An Open-Source Graph Database System. *Data Intell.* 5, 3 (2023), 560–610.
[73] Leyuan Wang and John D. Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. *CoRR* abs/2003.01527 (2020).
[74] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1 (2025), 39:1–39:27.

[75] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *ICDE*. IEEE, 1249–1260.

[76] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1 (2024), 60:1–60:29.

Received April 2025; revised July 2025; accepted August 2025