POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Subgraph isomorphism acceleration on FPGAs using High-Level Synthesis

Supervisors

Prof. Luciano Lavagno

Prof. Mihai Lazarescu

Candidate

Roberto Bosio

December 2022

Summary

Subgraph isomorphism (or subgraph matching) is a well-known NP-hard problem that consists of searching all the distinct embeddings of a query graph in a large data graph. It has a wide range of applications, almost in all the domains in which graph patterns reveal valuable information, especially in social network analysis, chemical compound search, and computer-aided design.

Due to the relevance of the problem, starting from 1970, several algorithms have been proposed, most of which adopt a backtracking approach by recursively mapping query vertices to data vertices. However, due to the intrinsic properties of graph computations, such as irregular communication patterns and little spatial and temporal locality, these algorithms cannot be easily accelerated in hardware.

An alternative approach to address the problem consists of seeing the query graph as a multiway join between relations, which are the edges, and where the vertices are the attributes. In recent years, the database community has proposed new join algorithms, typically called Worst-Case Optimal Join (WCOJ) algorithms, which have the property of bounding the number of intermediate results generated, in addition to showing an intrinsic concurrency.

Given these new results, the thesis studies the feasibility of accelerating the subgraph isomorphism by using a WCOJ approach, designing a kernel able to pre-process the graphs and evaluate the results all on FPGA. The proposed implementation works around a novel parameterized data structure based on hash tables, which has been designed to respect the complexity requirements of WCOJ algorithms as well as leave space for parameter optimization. The algorithm has been developed in C++ using Vitis™ HLS tool by Xilinx Inc.

Table of Contents

List of Tables

List of Figures

Acronyms

AGM
Atserias, Grohe, and Marx

CLB
Configurable Logic Block

DSP
Digital Signal Processing

EDA
Electronic Design Automation

FIFO
First In First Out

FPGA
Field Programmable Gate Array

HDL
Hardware Description Language

LUT
Look-Up Tables

WCOJ
Worst-case optimal join

Chapter 1

Background

1.1 Subgraph Isomorphism

Asking if two graphs are isomorphic or, in simpler words, if the two have the same "structure", is a fundamental problem in graph theory that arises in different scientific domains. It consists of finding a bijective function that maps one vertex set to the other so that, if two nodes are adjacent in one graph, their corresponding ones are also adjacent in the other graph.

A generalization of this problem is the so-called subgraph isomorphism, which asks if one graph contains a subgraph, a subset of vertices and edges, that is isomorphic to another graph. Practically, it can be seen as finding an instance of a graph (throughout the thesis referred to as *query graph*), in another one (*data graph*).

The subgraph isomorphism (or subgraph matching) problem is relevant in all the fields in which meaningful information is stored in graph patterns, it is used in computer-aided design as well as in social networks, Twitter for example, in their follower graph, search for diamond patterns to give recommendations to users [1].

1.1.1 Problem definition

The thesis focus on directed, connected, and vertex-labeled graphs, but all the techniques described can be extended to support other cases as edge-labeled graphs. A graph is represented by a set of vertices $V(g)$, a set of edges $E(g)$, and a labeling function $L_g : V(g) \to \Sigma$, where Σ is a set of labels. Given a *query graph* $q = (V(q), E(q), L_q)$ and a *data graph* $g = (V(g), E(g), L_g)$, a match, or embedding, is an injective function $f : V(q) \to V(g)$ such that:

1. $(v_1, v_2) \in E(q) \Rightarrow (f(v_1), f(v_2)) \in E(g)$
2. $\forall v \in V(q) : L_q(v) = L_g(f(v))$

The purpose of subgraph matching is to find all distinct instances of the *query graph* in the *data graph*, which is known to be NP-hard [2].

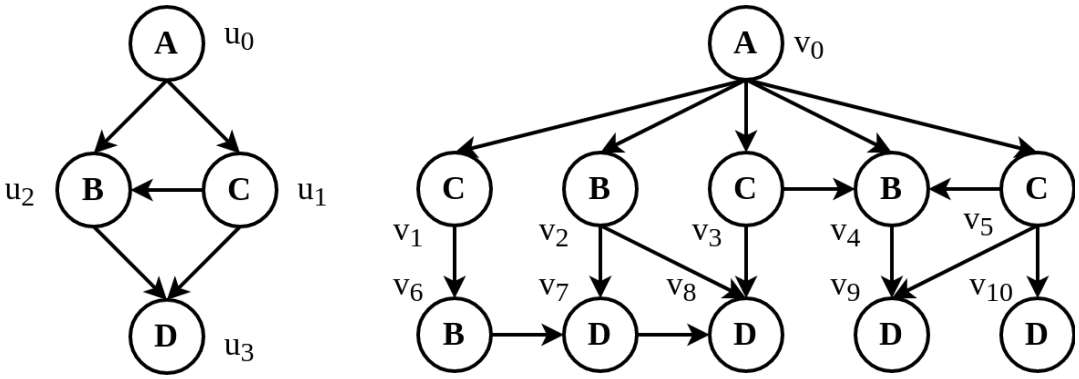

Figure 1.1: Example of query (left) and data (right) graphs.

1.2 Algorithms in literature

Starting in 1976, thanks to Ullmann's work [3], several algorithms have been developed to address the problem. Most of them use a backtracking approach, which consists in expanding a partial result by mapping query vertices to data vertices following a matching order, this method is referred to as *exploration-based*.

The other approach present in the literature is called *join-based* and it is more related to the database domain: it is possible to model the *query graph* as a relational join query, in which vertices and edges are respectively attributes and relations, such that evaluating the multi-way join retrieves all the embeddings.

Although these methods can both evaluate subgraph matching, the intrinsic differences make them suitable for different applications.

1.2.1 Exploration-based algorithms

State-of-the-art exploration-based algorithms are suitable to evaluate large queries, up to tens of vertices, in medium-sized datasets in the order of thousands to millions of vertices.

Several approaches have been tried in the years, and, although does not exist an algorithm able to outperform the others with any query, it has been shown that pre-processing the graphs to build auxiliary data structures performs better than directly exploring the *data graph* to enumerate the embeddings [4].

Pre-process

All the latest algorithms start by constructing a supplementary data structure that contains candidates set for each query vertex, as well as the edges between the candidates. This approach is useful to reduce the search space and filter out data vertices that are known to be not part of any solution. Some of the filters used to understand if a data vertex could be a candidate for a query vertex are:

1. vertex $v_i \in V(g)$ is a possible candidate for $u_i \in V(q)$ if v_i's label is equal to the label of u_i and the degree of v_i is bigger than u_i's degree.

2. vertex v_i is a possible candidate for u_i if, for each label in the neighborhood of u_i, the number of adjacent vertices to v_i with that label is bigger than the one adjacent to u_i

3. vertex v_i is a possible candidate for u_i if the intersection between the neighborhood of v_i and all the candidate sets of the vertices adjacent to u_i is not empty.

The third rule affirms something that is intuitive but not so trivial to be grasped, that is: if two vertices are connected by an edge in the *query graph*, then their respective candidate should be also connected in the *data graph*. Generalizing this idea, given a query vertex u and all the adjacent vertices u_n with their mapped data vertices v_n, the candidates for u must be adjacent to all the vertices v_n, in other words, all the possible embeddings for u are in the intersection between the neighbors of v_n. Thus, it is possible to retrieve candidate sets starting from the adjacent ones, and, in fact, state-of-the-art algorithms use this idea to build the auxiliary data structure.

Pruning the candidate sets as much as possible is fundamental to achieve better performances later in the computation, thus the algorithms spent a good amount of time refining these data structures. Figure 1.2 represents the candidate sets for

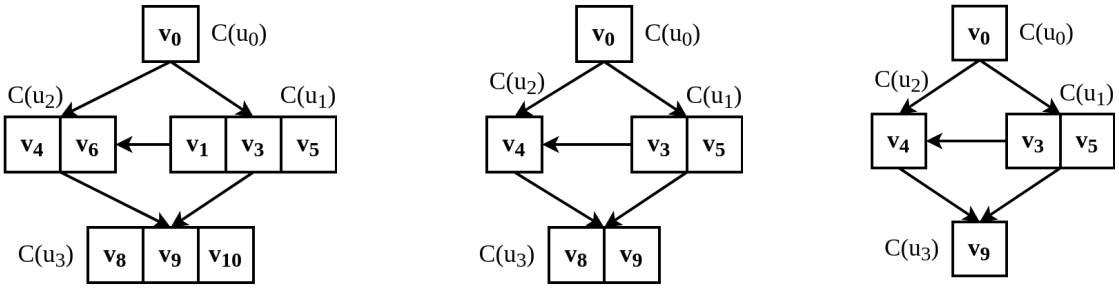

Figure 1.2: Refinement of candidate sets using filters.

the example in figure 1.1 after applying the filters described above. In particular, on the left it has been applied only the label and degree filter; then using the second filter is possible to remove the vertex v_6 from $C(u_2)$ since it has not an adjacent node labeled A, and vertex v_1 which miss in its neighborhood a node labeled D. The figure in the middle represents the candidate sets after the first two filters. Applying the third filter, following the order u_0, u_1, u_2, u_3, node v_8 is removed, since the intersection between its neighborhood $\{v_2, v_3, v_7\}$ and $C(u_2)$ is empty, has shown in the right figure. It is worth noticing that with another round of the third filter it would be possible to remove also node v_3, this is the reason why state-of-the-art algorithms do more than one step to refine the auxiliary data structure.

Query vertex order

The order used to match query to data vertices impacts the final algorithm's performance. Evaluate soon in the matching order the more selective query vertices will reduce the search space for the following nodes as well as lowering the size of intermediate results, speeding up the enumeration process.

Unfortunately, there is no exact method able to find the best search order in a reasonable amount of time, which led to several heuristics being developed over the years. State-of-the-art algorithms try to build a cost model based on the selectivity of each node, in general, using information like his degree and the size of his candidate sets. Although in literature there are several examples of cost models, there is still room for improvements [4].

Enumeration

Given the matching order and the auxiliary data structure, the enumeration procedure is in charge of computing the embeddings. State-of-the-art algorithms use a backtracking approach, and even if they have some differences in the way they enumerate the results, the behavior can be captured in a generic procedure.

In particular, for each local candidate of the current query vertex in the matching order, the partial solution is updated with the new pair query vertex - data vertex, then a recursive call is done on the next node in the order, and in the end, the previously added mapping is removed from the solution in a backtracking fashion.

Differences only arise in the computation of the local candidates, i.e. the candidates for the current query vertex given the already computed partial mapping, because of the distinct data structures computed in the pre-processing phase.

As an example, in case of no pre-processing, local candidates must be searched in the *data graph* between the neighbors of already matched vertices, instead if supported by an auxiliary data structure local candidates can be searched inside its respective candidate set.

Algorithm 1 Enumeration procedure

```
1: procedure Recursive_enumeration(i, M)
2:     if i ≥ |V(q)| then
3:         output M
4:         return
5:     end if
6:     u ← Current query vertex
7:     C_l ← Local candidates
8:     for all v in C_l do
9:         M ← M ∪ (u, v)
10:        Recursive_enumeration(i + 1, M)
11:        M ← M \ (u, v)
12:    end for
13: end procedure
```

1.2.2 Join-based algorithms

State-of-the-art join-based algorithms are suitable to evaluate small queries in large datasets which are typically in the order of millions to hundreds of millions of vertices.

In the last years, researchers have shown that a subgraph query can be seen as a self-join query in which every edge is a relation [5]. Every relation holds the *data graph* edges that are congruent with the direction and the nodes' labels relative to the query edge. In other words, the relations contain the candidates for every specific edge of the *query graph*. Evaluating the multi-way join between the relations finds all the instances of the query in the *data graph*.

Since the thesis focuses specifically on subgraph isomorphism, in the next sections it is always explained what the various operations in the database domain represent in the graph one.

Useful definition

This section employs the symbols used by Todd L. Veldhuizen in his work [6]. In relational terminology, *projection* (represented by the symbol π) means taking a subset of the relation's columns respecting given criteria, while *selection* (σ) means taking the rows respecting the condition. Given the relation $R(u_1, u_3)$, the set representing $\pi_{u_1}(R)$, will be referred to as $R(u_1, \_)$: in the specific case of graphs, the set is composed by all the data vertices labeled $L(u_1)$ having an edge compatible with the one between u_1 and u_3. The set $\pi_{u_3}(\sigma_{u_1=v_5}(R))$ will be referred to as $R_{v_5}(u_3)$: it contains all the data vertices labeled $L(u_3)$ having an edge with the same direction as the one between u_1 and u_3 and connected to v_5.

u_1	u_3
v_3	v_8
v_5	v_9
v_5	v_{10}

R

u_1	u_3
v_3	v_8
v_5	v_9
v_5	v_{10}

$\pi_{u_1}(R)$

u_1	u_3
v_3	v_8
v_5	v_9
v_5	v_{10}

$\pi_{u_3}(\sigma_{u_1=v_5}(R))$

Figure 1.3: Example of projections and selections on a relation

Edge-at-a-time example

Taking the triangle query in figure 1.4 it is possible to extract three instances of the same relation $R(u_i, u_j)$ which are $M(u_0, u_1)$, $N(u_1, u_2)$, $S(u_2, u_0)$. Joining the tables M and S on the attribute u_0 will have as a result a table P storing the information of all the data vertices which have (1) the same label as u_0, (2) an outgoing edge to a vertex with u_1's label and (3) an incoming edge from a vertex with u_2's label. Joining P with N will discard all the entries not respecting the edge between u_1 and u_2, returning the final result. This approach is called *edge-at-a-time*: it uses a sequence of binary joins, each one adding a new edge to the intermediate results.

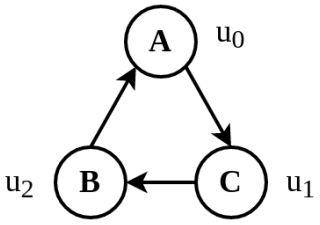

Figure 1.4: Triangle query

Vertex-at-a-time example

The other approach is called *vertex-at-a-time*, the algorithm below shows the pseudo-code to solve the triangle query, taken from the Hung Q. Ngo work on worst-case optimal join algorithms [7]. Again the three same relations $M(u_0, u_1)$, $N(u_1, u_2)$, $S(u_2, u_0)$ are extracted.

At line 1, the intersection generates the set P_{u_0} containing all the data vertices labeled A with both an outgoing edge to a vertex B and an incoming edge from a vertex C. Starting from a given vertex v_0, line 3 computes the set $P_{u_1}^{v_0}$ which contains all the data vertices labeled B with (1) an incoming edge from v_0 and (2) an outgoing edge to a C vertex. Ultimately, line 5 finds the C vertices adjacent to v_0 and v_1, generating valid embeddings.

Algorithm 2 Vertex-at-a-time algorithm for triangle query

1: $P_{u_0} \leftarrow M(u_0, \_) \cap S(\_, u_0)$
2: **for all** $v_0 \in P_{u_0}$ **do**
3: $\quad P_{u_1}^{v_0} \leftarrow M_{v_0}(u_1) \cap N(u_1, \_)$
4: $\quad$ **for all** $v_1 \in P_{u_1}^{v_0}$ **do**
5: $\quad\quad P_{u_2}^{v_0,v_1} \leftarrow N_{v_1}(u_2) \cap S_{v_0}(u_2)$
6: $\quad\quad$ **for all** $v_2 \in P_{u_2}^{v_0,v_1}$ **do**
7: $\quad\quad\quad$ Add (v_0, v_1, v_2) to results
8: $\quad\quad$ **end for**
9: $\quad$ **end for**
10: **end for**

Query vertex order

As already discussed for exploration-based algorithms, the query vertex order used to evaluate the subgraph matching plays a fundamental role in the final performance of the algorithm. In the case of join-based algorithms, choosing a different order in the query vertices translates to applying a different join plan. Again, there is no optimal solution, so different heuristic cost models are used in the literature to estimate the goodness of a QVO.

Approaches: comparison

A thorough comparison between binary and multi-way join is the central topic of the next section, nonetheless, some conclusions can be drawn from the above introduction to join-based algorithms. In the example, the intermediate result from *edge-at-a-time* approach $P = Q \bowtie S$ contains already all the instances of the query in the *data graph* plus some entries that will be removed in the last join. This deduction implies that the size of the intermediate result will be bigger or equal to the size of the final one, which is not optimal. Although the above example does not show it, since in the triangle query the number of vertices is equivalent to the number of edges, the join plan between *edge-at-a-time* and *vertex-at-a-time* are different. Figure 1.5 shows them for the diamond-X query, representing on the left the binary join plan, and on the right the worst-case optimal one.

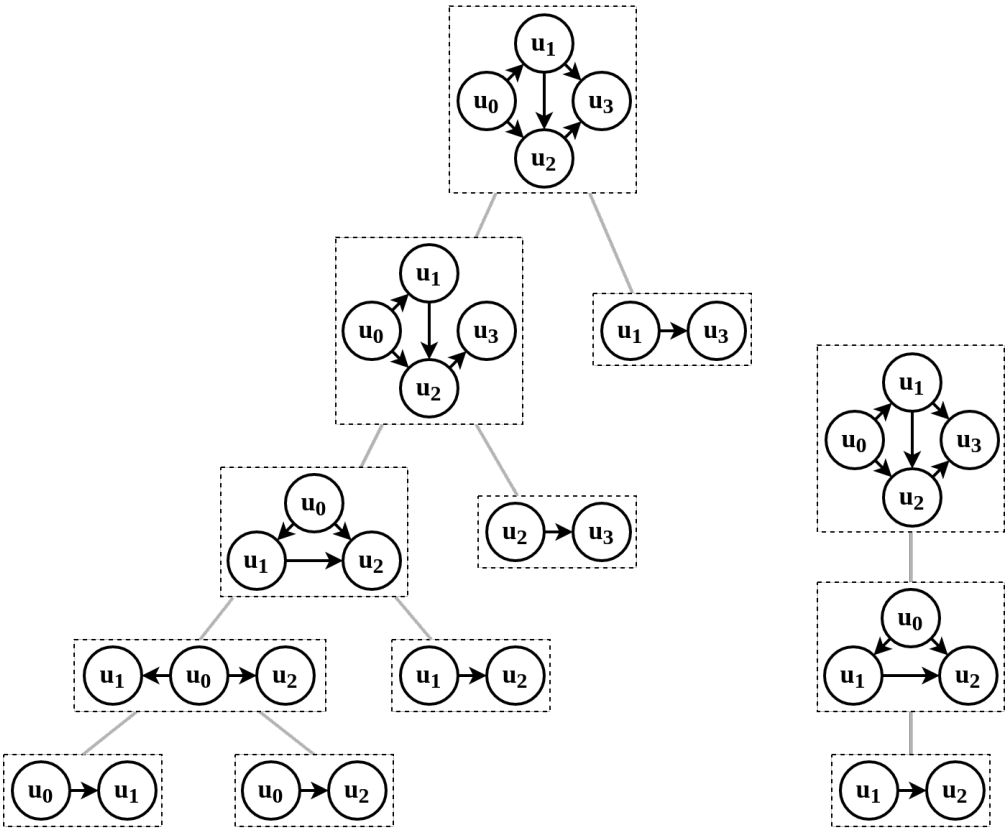

Figure 1.5: Join plans

Approaches: state-of-the-art

State-of-the-art algorithms, such as Graphflow [8], have not wholly removed binary join in favor of multi-way join inside their join plans, but instead have tried to build hybrid plans mixing the two approaches. The reasons are: (1) binary join plans are optimized by decades-long research, and (2) binary join plans are suboptimal only for cyclic queries.

1.3 Worst-case optimal join

The computation of relational join is a well-studied problem in the database community with decades of research behind it, resulting in finely optimized solutions currently used in modern commercial tools. Nonetheless, recent studies have shown that, for particular categories of queries, traditional pair-wise join plans are suboptimal no matter the join order selected, opening space for algorithms able to be theoretically optimal even in worst-case scenarios [9]. This section introduces

the algorithm on which is based the implementation designed in the thesis work mainly by summarizing the results of H. Q. Ngo, C. Ré and A. Rudra [7].

1.3.1 AGM bound

Query output size estimation is central in join processing due to the fact that join queries are typically expected to evaluate the result in time asymptotic to the output size. Atserias, Grohe, and Marx (AGM henceforth) recently derived a non-trivial bound on the maximum output size of a full conjunctive query based on the input relations' size.

Useful definitions

A conjunctive query is a class of queries that only contains selections, projections, and joins. The size of a query q, i.e the number of entries, is represented as $|q|$.

A hypergraph is a generalization of standard graphs in which edges can connect more than two vertices and it is represented by the pair $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, in which $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of hyperedges.

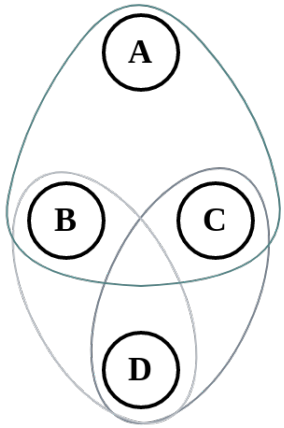

Figure 1.6: Example of a hypergraph.

A query q can be modeled as a hypergraph in which the vertices $\mathcal{V}$ are the set of attributes and each hyperedge $F \in \mathcal{E}$, which is a subset of vertices, defines a relation R_F. The set of edges covering a node v is defined as $v_\mathcal{E} \subseteq \mathcal{E}$.

The *fractional edge cover* of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a vector x of weights assigned to each hyperedge F such that $\forall v \in \mathcal{V}, \sum_{F \in v_\mathcal{E}} x_F \geq 1$. In other words for each node $v \in \mathcal{V}$, the sum of the edge's weight covering v is bigger or equal to 1.

The *minimum fractional edge cover* is the solution to the optimization problem:

$$\begin{aligned} \rho^*(H) = \min \quad & \sum_{F \in \mathcal{E}} x_F \\ \text{s.t.} \quad & \sum_{F \in v_{\mathcal{E}}} x_F \geq 1 \qquad \forall v \in \mathcal{V} \\ & x_F \geq 0 \qquad\quad \forall F \in \mathcal{E} \end{aligned}$$

Output size bound examples

A simple join query as $q(a, b, c) = R(a, b), S(b, c)$ has a maximum output size of $|R| \cdot |S|$. An example of a query that reaches this bound is

$$R = \{a_0, ..., a_m\} \times \{b_0\}$$
$$S = \{b_0\} \times \{c_0, ..., c_m\}$$

In this case, the join behaves as a cartesian product.

Consider the triangle query $q(a, b, c) = M(a, b), R(b, c), S(c, a)$. A first straightforward bound on the output size could be $|M| \cdot |R| \cdot |S|$. But as already observed in the last paragraph of subsection 1.2.2 the join result of two of the three relations is already a superset of the final result, thus an even better bound is given by $\min\{|M| \cdot |R|, |R| \cdot |S|, |M| \cdot |S|\}$. Since evaluating this query means counting all the triangles in a graph, which is a relevant problem, the researchers found another bound for the maximum query output size which is $\sqrt{|M| \cdot |R| \cdot |S|}$. A simple and beautiful demonstration from H. Q. Ngo, C. Ré, and A. Rudra [7] on how to retrieve this bound without using the AGM inequality is given in appendix A. It is interesting to understand which of the two is the tightest one.

AGM inequality

Given a conjunctive query q with n relations, AGM demonstrates that for every fractional cover x, the following inequality holds:

$$|q| \leq \prod_{j}^{n} |T_j|^{x_j}$$

Since there could be several edge covers, it is relevant to compute the one that minimizes the quantity $\prod_j^n |T_j|^{x_j}$. After doing a logarithmic transformation, the new optimization problem to find the minimum cover edge became:

$$\begin{aligned} \min_z \quad & \sum_{j} \log(|T_j|) \cdot x_j \\ \text{s.t.} \quad & \sum_{F \in v_E} x_F \geq 1 \qquad \forall v \in V \\ & x_F \geq 0 \qquad\quad \forall F \in E \end{aligned}$$

AGM bound examples

- Considering again the triangle query and relations sizes $|M| = |R| = |S| = N$, the optimization problem can be written as follow:

$$\begin{aligned} \rho^* = \min_x \quad & \log(N) \cdot (x_1 + x_2 + x_3) \\ \text{s.t.} \quad & x_1 + x_2 \geq 1 \\ & x_2 + x_3 \geq 1 \\ & x_1 + x_3 \geq 1 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

  The minimum is achieved when $x_1 = x_2 = x_3 = 0.5$ which results in $\rho^* = log(N) \cdot 1.5$. The AGM inequality in this case became

$$|q| \leq |M|^{0.5} \cdot |R|^{0.5} \cdot |S|^{0.5} = \sqrt{|M| \cdot |R| \cdot |S|} = N^{\frac{3}{2}}$$

- Changing the input sizes to $|M| = |R| = N_1$ and $|S| = N_2$ with $N_2 \gg N_1$ then:

$$\begin{aligned} \rho^* = \min_x \quad & \log(N_1) \cdot (x_1 + x_2) + \log(N_2) \cdot x_3 \\ \text{s.t.} \quad & x_1 + x_2 \geq 1 \\ & x_2 + x_3 \geq 1 \\ & x_1 + x_3 \geq 1 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

  The minimum $\rho^* = 2\log(N_1)$ with $x_1 = x_2 = 1$ and $x_3 = 0$, giving a size bound

$$|q| \leq |M|^1 \cdot |R|^1 \cdot |S|^0 = \min\{|M| \cdot |R|, |R| \cdot |S|, |M| \cdot |S|\} = N_1^2$$

These two examples show that the two previously given output size bounds for the triangle query are equally valid.

1.3.2 Generic-Join algorithm

Since for the triangle query with relations equally sized the maximum output size is equal to $N^{\frac{3}{2}}$, it is interesting to understand if traditional pair-wise join plans are able to always run this query in time $\Omega(N^{\frac{3}{2}})$.

Motivating example

The triangle query $q(a, b, c) = M(a, b), R(b, c), S(c, a)$ with relation defined as

$$\begin{aligned} M &= \{a_0\} \times \{b_0, ..., b_m\} \quad \cup \quad \{a_0, ..., a_m\} \times \{b_0\} \\ R &= \{b_0\} \times \{c_0, ..., c_m\} \quad \cup \quad \{b_0, ..., b_m\} \times \{c_0\} \\ S &= \{a_0\} \times \{c_0, ..., c_m\} \quad \cup \quad \{a_0, ..., a_m\} \times \{c_0\} \end{aligned}$$

and sizes $|M| = |R| = |S| = 2m - 1$. The three possible join plan choices are represented in figure 1.7.

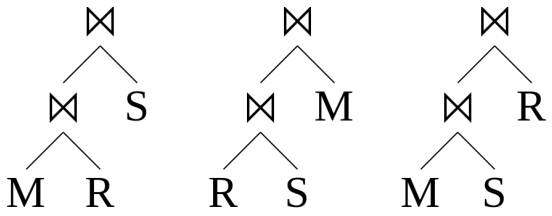

Figure 1.7: Binary join plans for the triangle query

Evaluating all of the three possibilities, the intermediate results P are:

$$\begin{aligned} M \bowtie R &= \{a_0\} \times \{b_1, ..., b_m\} \times \{c_0\} \quad \cup \quad \{a_0, ..., a_m\} \times \{b_0\} \times \{c_0, ..., c_m\} \\ R \bowtie S &= \{a_0\} \times \{b_0\} \times \{c_1, ..., c_m\} \quad \cup \quad \{a_0, ..., a_m\} \times \{b_0, ..., b_m\} \times \{c_0\} \\ M \bowtie S &= \{a_1, ..., a_m\} \times \{b_0\} \times \{c_0\} \quad \cup \quad \{a_0\} \times \{b_0, ..., b_m\} \times \{c_0, ..., c_m\} \end{aligned}$$

Due to the symmetry used to build the query, all the partial results have the same size, in particular, the sets before the union have size m while the second ones have size $(m + 1)^2$, and, since there are no overlapping entries, the total size is $P = (m + 1)^2 + m \in \Theta(m^2)$. The final result instead is:

$$\begin{aligned} Q = &\{a_0\} \times \{b_0\} \times \{c_1, ..., c_m\} \\ &\cup \{a_0\} \times \{b_1, ..., b_m\} \times \{c_0\} \\ &\cup \{a_0, ..., a_m\} \times \{b_0\} \times \{c_0\} \end{aligned}$$

which has a size equal to $3m + 1 \in \Theta(m)$. Since join queries generally take an amount of time asymptotic to the result size, this example shows that, for a particular category of queries, pair-wise join plans are suboptimal. This is the reason why a new category of join algorithms has been researched and developed, trying to reach optimality even in the worst case.

Algorithm

Generic-Join is a worst-case optimal join algorithm, in the sense that it is able to always run in linear time, up to a log factor, with respect to the output size of the worst-case scenario.

Algorithm 3 Generic-Join($\bowtie_{F \in \mathcal{E}} R_F$)

Input: Query $\mathcal{Q}$, hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$

1: $\mathcal{Q} \leftarrow \varnothing$
2: **if** $|\mathcal{V}| = 1$ **then**
3: return $\cap_{F \in \mathcal{E}} R_F$
4: Pick I arbitrarily such that $1 \leq |I| < |\mathcal{V}|$
5: $L \leftarrow$ Generic-Join($\bowtie_{F \in \mathcal{E}_I} \pi_I(R_F)$)
6: **for** every $t_I \in L$ **do**
7: $\mathcal{Q}[t_I] \leftarrow$ Generic-Join($\bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes t_I)$)
8: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{t_I\} \cup \mathcal{Q}[t_I]$
9: **return** $\mathcal{Q}$

For the sake of clarity, the Generic-Join algorithm is described from the subgraph matching point of view, in particular, the hypergraph in input is a simple graph, in the sense that edges are not hyperedges and so they can connect only two vertices. This simplification also implies that relations R have only two attributes: the source and the destination vertices of the query edge they represent.

Line 4 chooses a set I which is a subset of the query vertices V, for simplicity, during this paragraph it is always picked a set with cardinality $|I| = 1$, i.e a single vertex u.

Line 5 computes the candidate set for the vertex u by doing a recursive call on the query $\bowtie_{F \in \mathcal{E}_I} \pi_I(R_F)$ which is composed by the projection on u of the relations in which u is an attribute. In graph terms, the query is formed by the sets of data graph vertices which are potential candidates for u in the edge represented by each relation.

The loop at line 6 iterates through the candidates t_I of u, and, for each one, compute the Generic-Join on a query $\bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes t_I)$, where J is the set of vertices $V \setminus I$. Differently from the recursive call at line 6, the term $R_F \ltimes t_I$ is doing a selection on $u = t_I$ in the relations in which u is involved, which means removing from the relations all the candidates for vertices adjacent to u which are not adjacent to t_I. Then the algorithm will recursively compute the embeddings for the next query vertex chosen, extending the partial solution by one embedding, and so on.

The algorithm 2 employed to compute the triangle query can be seen as a

specialization of the Generic-Join in which $I = \{u_0, u_1\}$.

Complexity and consideration

Considering a query, its hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$, and a fractional edge cover x of $\mathcal{H}$, the time complexity of Generic-Join is $\tilde{\mathcal{O}}(mn \prod_{F \in \mathcal{E}} |R_F|^{x_F})$ in which the $\tilde{\mathcal{O}}$ hides a potential log factor. To achieve the mentioned complexity, it is required:

- The set intersection $\cap_{F \in \mathcal{E}} R_F$ to run in time complexity $\tilde{\mathcal{O}}(m \cdot \min|R_F|)$
- A global attribute order and pre-indexed relations in such a way that retrieve $R_F \ltimes t_I$ is possible at max in logarithmic time.

It is worth noticing that theoretical optimality does not mean always better performances with respect to the pair-wise join, in fact, much of the Generic-Join computation time is dependent on the vertex query order.

1.4 Field programmable gate array

FPGAs are programmable integrated circuits formed by several arrays of logic elements, called CLBs, which are able to compute arbitrary functions. CLBs are typically composed of a LUT, a full adder, and a flip-flop, and the interconnections between them are programmable, as well as the connection to the I/O blocks. Modern FPGAs integrate also specific blocks to speed-up common operations and save some CLBs, like DSPs, multipliers, and embedded memories.

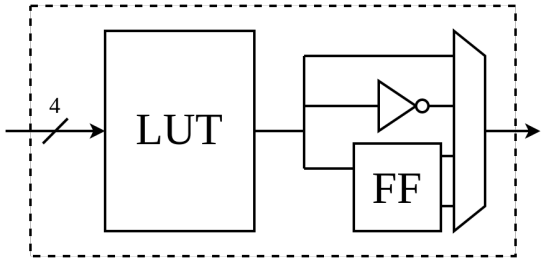

Figure 1.8: Example of a simple CLB

1.4.1 High-Level Synthesis

FPGAs can be configured via a Hardware Description Language like VHDL or Verilog, however writing HDL code is generally complex and require a lot of time. High-Level Synthesis is a technique aimed to reduce the distance between

software and hardware development, by retrieving an HDL model from a high-level programming language like C++. The tradeoff of using HLS is to lose some low-level optimization in favor of a much faster developing phase. The HLS tool used in the thesis is Vitis™ HLS 2021.2 by Xilinx [10].

HLS steps

- The first steps consist in writing the high-level implementation of the design. Not all the C++ language is available to the developer, for example, all the system calls to the operating system are not supported.
- After finishing the code development is fundamental to verify the software functionality: the program is verified against a testbench which is in charge of comparing the result produced by the implementation to the expected one.
- Once passed the software functionality test, it is possible to synthesize the code to obtain an RTL model.
- The last step is to verify that the RTL model behaves in the same way as the software description by doing a co-simulation.

HLS optimizations

HLS tool offers several optimization directives to improve the performances of the design, exploiting the hardware advantages:

- The memory optimizations allow configuring how the arrays are stored in memory, giving to developer the ability to optimize the resources based on the actual usage of the array. As an example, an array can be partitioned to increase the read and write ports or can be reshaped to read multiple elements in a single transaction.

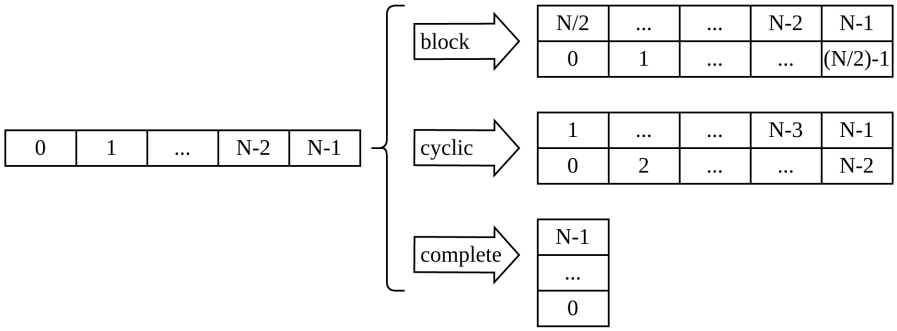

Figure 1.9: Example of reshape optimization.

- The unroll optimization allows to execute in parallel all the loop iterations, or a fraction of them, by creating several copies of the operation in the loop body. This optimization can be used when there are no data dependencies between the loop iterations.

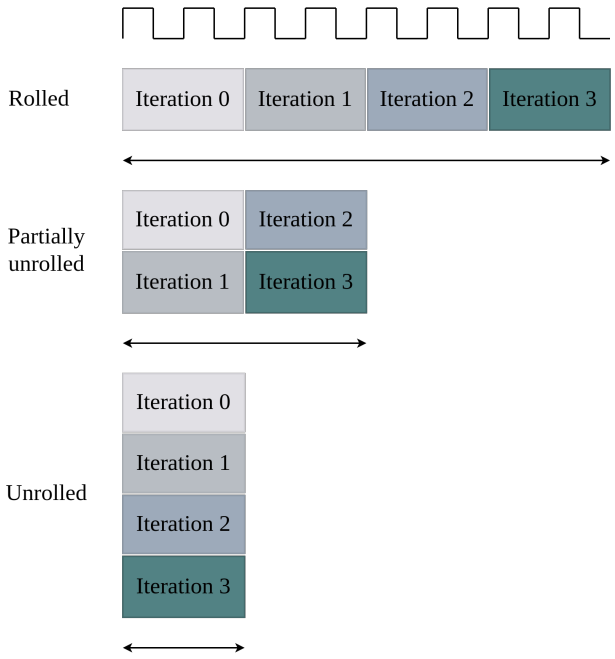

Figure 1.10: Example of query unrolling optimization.

- The dataflow optimization tries to exploit task-level parallelism by inferring data dependencies between them. In case of no dependency (or partial), two tasks can run in parallel improving the design throughput. This optimization comes at a cost of a small hardware overhead needed to implement the channels between the tasks.

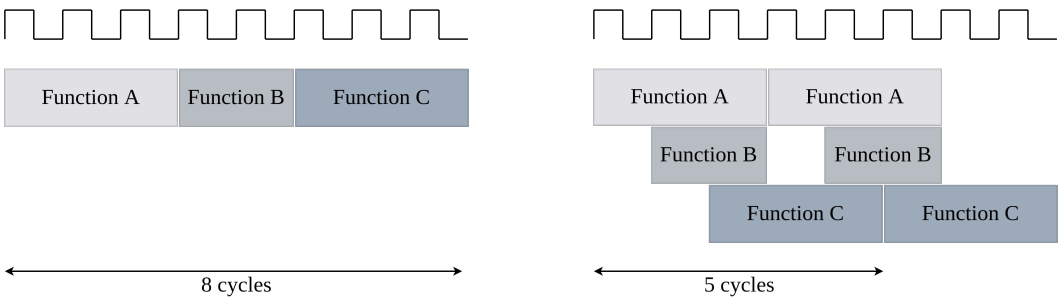

Figure 1.11: Example of dataflow optimization.

Chapter 2

Motivation

In recent years there has been a rising interest in accelerating various graph algorithms on FPGAs. However, even if subgraph isomorphism is a well-studied problem in literature with several algorithms proposed, most of the state-of-the-art solutions are targeting CPUs and GPUs, without considering FPGAs as a viable option (with the unique exception of FAST [11]). The reason behind this choice could be searched in the intrinsic properties of graph elaboration: most of the time is spent accessing data with little or no temporal and spatial locality, rather than processing it [12].

This research's deficiency in the FPGAs direction is the main motivation behind the thesis work.

2.1 Contributions

The proposed solution tries to bring new ideas to address the subgraph isomorphism on FPGA. Differently from the already-mentioned FAST algorithm, which is implementing an exploration-based method and a pre-processing phase on the CPU, the thesis work approaches the problem from another point of view, introducing:

- A new implementation with pre-processing and enumeration all on FPGA.
- A Generic-Join based algorithm executed with four fully pipelined stages.
- A parametrized data structure to store the *data graph* having constant access time while not occupying a large portion of memory.

Chapter 3

Proposed implementation

This section thoroughly describes the proposed solution in a top to bottom fashion, by dissecting the general structure of the kernel. Particular emphasis is given to the motivations behind the design choices as well as the compromises accepted during the development.

3.1 Overview

The kernel is theoretically split into two non-overlapping phases:

- The pre-processing: in charge of building all the data structure needed in the future for the subgraph matching. It is itself formed by three non-overlapping phases for building table descriptors, counting collisions, and storing the edges.
- The multiway join: in charge of evaluating the query against the *data graph*, it is formed by four stages working in parallel connected in a loop.

The multiway join phase cannot start until the pre-processing has finished elaborating the last *data graph* edge since the multiway join could not build consistent partial results without completed data structures.

Kernel interface

The kernel is interfaced with the host through four input streams and one output stream. The four input streams are used to communicate edges, for both query and data graphs, which are composed of the source and the destination vertex and their respective labels. The source vertex stream is also communicating the query vertex order. The output stream is the channel dedicated to the final embeddings, streaming the result one vertex id at-a-time in a sequential fashion.

The second part of the top function interface is composed of a pointer to the whole memory, which has been allocated by the host and will be later handled by the kernel itself. While virtually is only one array, in the actual parameters are present five identical arrays all pointing to the same memory location, but connected with distinct AXI4 memory-mapped interfaces. The necessity of this arrangement comes from the fact that multiple functions in the multiway join are reading from the same array concurrently, which is not possible with one unique pointer.

Inputs format

The kernel does not put constraints on the format of the data graphs, edges can be shuffled with vertex ids not starting from 0. However, the query vertices must start from 0, with again no constraint on the order of the edges.

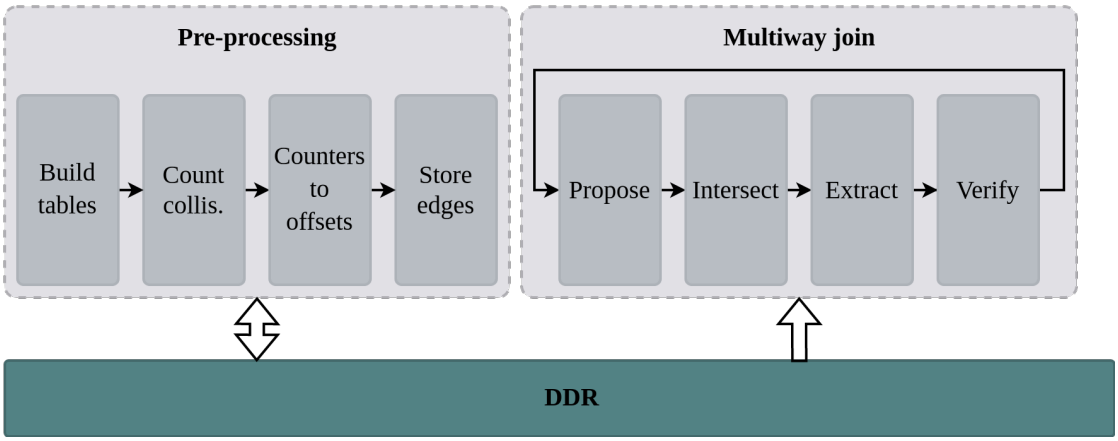

Figure 3.1: Producer-consumer point of view of the kernel

3.2 Pre-process data

The pre-processing phase is conceptually divided into four macro non-concurrent functions. Before describing them, it is worth introducing some concepts extensively used in the following paragraphs.

Useful definition

In this section, the word *table* defines an instance of a relation, as the one shown in figure 1.3. Each query edge corresponds to a table containing all the compatible data graph edges. A table descriptor is a structure used to store the details defining

a specific table, which are: the source vertex label, the destination vertex label, and the direction of the table. The two query vertices considered inside a table take different names based on their position in the QVO, the first one in the order is called *indexing vertex*, and the other one *indexed vertex*. The notation $h(v_0)$ stands for the hash value of v_0 while h_{max} stands for the maximum possible value of the hash.

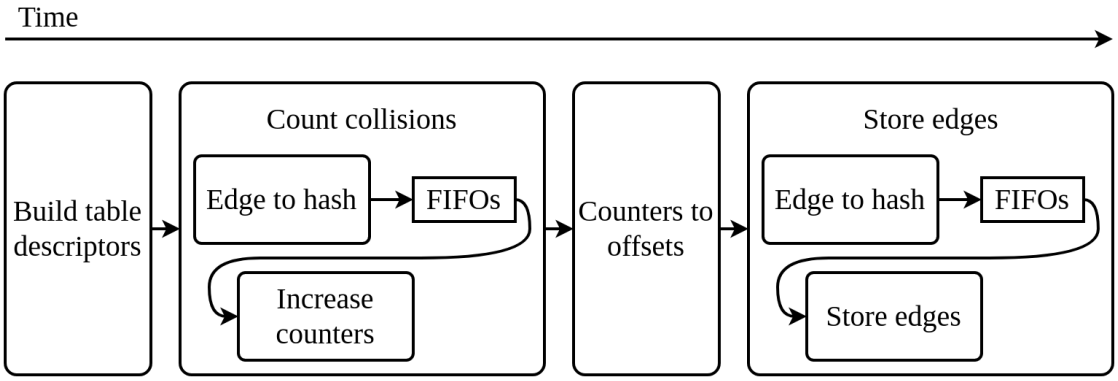

Figure 3.2: Pre-process phase

3.2.1 Data structure

The implemented solution is composed of two parts: a rectangular matrix-like data structure storing offsets and the actual edges of the table. The offsets are used to retrieve specific portions of edges, or, by comparing them, understand if a specific edge is absent. As a result of the pre-processing algorithm, the array of edges is sorted based on the hash values of its vertices.

General description

The relational operations that the data structure must efficiently support are the ones utilized in the Generic-Join algorithm. Specifically given a table $R(u_0, u_1)$, a QVO $\{u_0, u_1\}$, a vertex v_0 candidate for u_0, and a vertex v_1 candidate for u_1:

- $\pi_{u_0}(R)$ is done by accessing directly the edges and selecting only the indexing vertices.
- $\pi_{u_1}(\sigma_{u_0=v_0}R)$ is accomplished by computing $h(v_0)$, using it to retrieve the range of v_0 edges, and accessing the edges selecting only the indexed vertices.

Detailed description

Every table has, other than a table descriptor, a global memory address at which starts the matrix and another address at which starts the array of edges. Given a specific offset obtained by the row index $h(v_0)$ and column index $h(v_1)$, it is possible to sum this number to the start address of the edges to obtain the location in memory of the edges compatible with $h(v_0)$ and $h(v_1)$. This set is composed of all the edges which share the same indexing vertex hash and indexed vertex hash (without hash collisions this set would be at max composed by one element).

Read To be specific, every offset represents the end address of each set, and so: read the set of edges $\sigma_{h(u_j)=h(v_1) \wedge h(u_i)=h(v_0)}R$ requires to read the end offset as specified above and read the start offset by indexing the matrix with $h(v_0)$ and $h(v_1) - 1$. For clarity of exposition, it is not considered the case in which $h(v_0) = 0$ or $h(v_1) = 0$. In the same way reading the set of edges $\sigma_{h(u_i)=h(v_0)}R$ requires two accesses in matrix, specifically at $h(v_0)$, h_{max} and $h(v_0) - 1$, h_{max}.

Check It is worth noticing that if the two addresses are equal, the set is empty. Considering this idea, it is possible to check the presence of an element in a set: testing if $v_1 \in \pi_{u_1}(\sigma_{u_0=v_0}R)$ can be done by retrieving the two delimiters of the set $\sigma_{h(u_j)=h(v_1) \wedge h(u_i)=h(v_0)}R$ and comparing them. Due to collisions, however, false positives are possible, but not false negatives. Since this property is at the base of the set intersection, to reduce the accesses in memory it has been chosen to spend the MSB of every offset to flag if a set is empty or not.

Space complexity The space complexity of the matrices is decided at compile time, by tuning the parameters defining the hash set cardinality. Reducing the hash bit-width will save space and reduce pre-processing time, however, it will also increase collisions, rising the time spent by the multiway join to detect the false positives produced by the set intersection.

Motivation to design choices

- Since no assumption is taken on the range of possible vertex ids, indexing a matrix with them is not feasible. Two solutions were evaluated at design time: translating them to the bounded set $\{0, |V|\}$, or using the hash of the vertex id as the actual index. Since the former solution requires keeping a dictionary, the latter is the one chosen, even because using hashes gives the freedom of choosing the cardinality of the set to which vertex ids are mapped.
- Since the degree of every vertex is normally much smaller than the number of vertices in the graph, keeping a square matrix would result in a waste of space,

in particular in this specific case where every table is storing only a fraction of the total graph edges. This is the main reason behind the decision of using two different cardinalities for the hash sets indexing rows and columns, with the latter smaller than the former, giving the matrix a rectangular shape.

- The main advantage of this implementation is the constant time required to execute most of the operations required by the multiway join, reducing the linear scanning of edges only to small sets sharing the same vertices' hash value.

3.2.2 Algorithm

The general algorithm can be thought of as an instance of the counting sort algorithm split on different arrays, one for each table, ordering their respective portion of edges.

Building table descriptors

The pre-processing phase starts by reading the QVO from the input streams since it is used to determine the direction of each table: if the source comes before the destination then the direction is normal, otherwise is inverted. Keeping this information is necessary to understand which vertex index the rows and which one index the columns of the table's matrix. The vertex indexing the rows is called the indexing vertex, while the one indexing the columns is referred to as the indexed vertex: this definition is the same given at the start of the chapter but seen from a more operative point of view.

After reading the QVO, instancing a class *QueryVertex* for each node, and saving the total number of query vertices, it is the turn of the query edges. For every edge, the first operation is to understand if a table descriptor with the same specification is already present, if so a new table is not necessary. Then the index of the table representing the current query edge is saved in the two *QueryVertex* involved, in a way that in the next steps it will be possible to retrieve for every query vertex all the tables in which it is involved.

Counting edges

At this point, the number of tables needed is known, thus is possible to compute the start address of each matrix, given their fixed occupation. The matrices are located one after the other in memory, detached from the arrays of edges that they refer to.

The first round of data edges is used to count the collisions inside the cells in the matrix. Even if at the end the matrix will contain offsets, at this stage it is used to

store counters. In particular, for each edge in the data graph, first is individuated the table in which it should be stored, then the vertices are used to compute the hash values, which consequently are exploited to locate the correct counter and increase it.

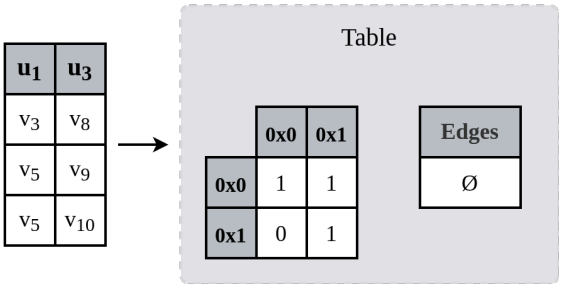

Figure 3.3: Example of table after the *Count collisions* phase.

In the figure above an example of a table at this point of the elaboration, given $h(v_5) = h(v_{10}) = 0$ and $h(v_3) = h(v_8) = h(v_9) = 1$. Now is possible to transform, for all the tables, the counters to offsets. Each counter should store the number of edges counted from 0 to that specific point, which can be done in linear time with the number of counters by employing an additional variable to store the total sum until that point. After this passage, the counters represent the starting address of each bucket of edges.

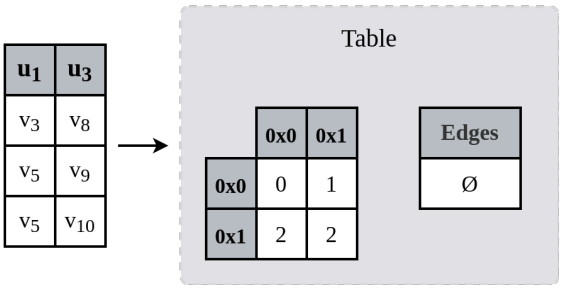

Figure 3.4: Example of table after the *Counters to offsets* phase.

Storing edges

After the counting edge phase, the number of edges that will be contained in every table is known, which is enough to compute the starting address of each edge array.

The second round of data edges is similar to the first one, this time however the offset is used to store the edge and only then increased. The result is a matrix containing the offsets representing the end address of each set of edges, and an

array of edges sorted on the hash value of the vertices (indexing vertex as first key, indexed vertex as second).

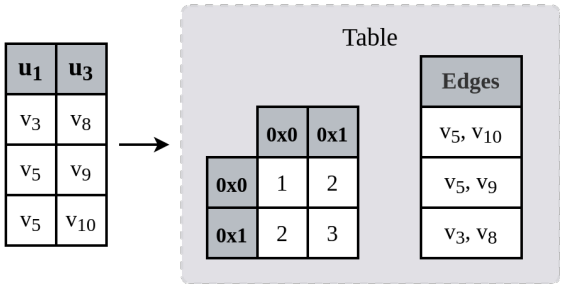

Figure 3.5: Example of table after the *Store edges* phase.

Motivation to design choices

This kind of hash table implementation requires reading two times the edges of the data graph, this is in order to handle collisions and use the right amount of memory space, given the fact that data is not changing in time. Other protocols for collision were evaluated during the design phase, for example, allocating a predefined amount of space for each bucket of edges. This solution would have required only one round of data edges but also the handling of an overflow area that stores the edges exceeding the fixed space. Then the operation of retrieving a set of edges, in some cases, would have comported searching in the overflow area. In the end, it is a trade-off between the performance of the pre-processing and the performance of the multiway join.

3.3 Multiway join

The multiway join algorithm proposed is based on Generic Join, expanding each partial embedding one vertex at-a-time and adopting a more hardware-friendly iterative process to compute the results.

The first line of the pseudo-code adds a void partial result, which is needed to start the process. Line 5 is the actual core of the algorithm, which is executed by four pipelined functions:

- Propose: find the set with minimum size in $\pi_u(R_u \ltimes p)$ and read it from memory.
- Intersect: do an approximative set intersection based on hash values.
- Extract: retrieve the vertex ids from the result of the intersection.

Algorithm 4 Multiway join algorithm

Input: QVO ϕ, tables R

1: $\mathcal{P} \leftarrow (\varnothing)$
2: **for all** $u \in \phi$ **do**
3: $\quad \mathcal{P}_u \leftarrow \varnothing$
4: $\quad$ **for all** $p \in \mathcal{P}$ **do**
5: $\quad\quad L_u \leftarrow \cap\, \pi_u(R_u \ltimes p)$
6: $\quad\quad \mathcal{P}_u \leftarrow \mathcal{P}_u \cup L_u$
7: $\quad$ **end for**
8: $\quad \mathcal{P} \leftarrow \mathcal{P}_u$
9: $\quad$ **if** $\mathcal{P} = \varnothing$ **then**
10: $\quad\quad$ break
11: $\quad$ **end if**
12: **end for**

- Verify: check the new partial result correctness and store it.

The algorithm terminates when all the vertices in the QVO have been considered or when the partial result FIFO goes empty (failed subgraph matching).

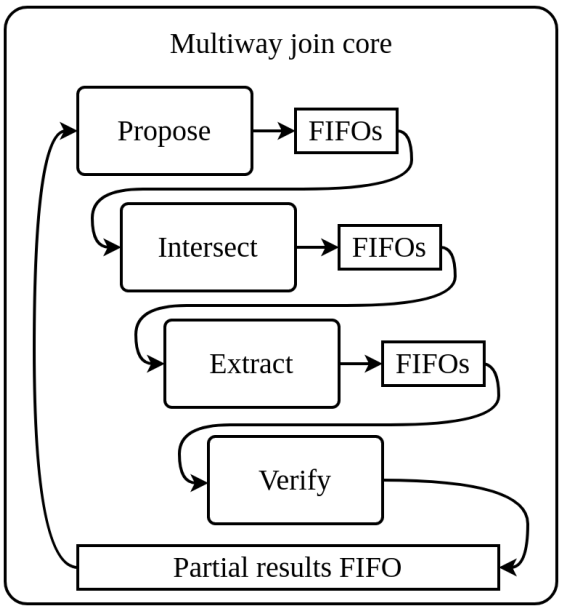

Figure 3.6: Multiway join pipeline.

3.3.1 Propose

The propose is composed of two functions executed in sequence: the first one is in charge of retrieving the set sizes involved in the set intersection while keeping track of the minimum one. Found the smallest set, the second function has to read it from the memory.

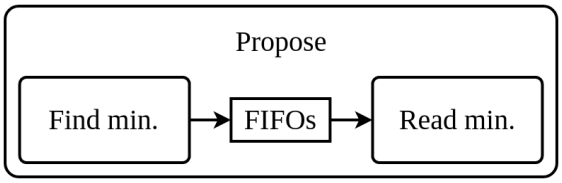

Figure 3.7: Propose pipeline.

Propose: find minimum set

This procedure is essential since WCOJ theory requires the set intersection to be proportional to the size of the minimum set, as stated in section 1.3.2. Since the set intersection is done between the hash values of the nodes, the compared sizes are relative to the set of hashes involved.

Given the current query vertex index u_i, it is possible to recover the tables in which it is involved by looking at its respective instance of the class *QueryVertex*. Retrieving the size of the sets in which the vertex is indexing the table, i.e $|\pi_{u_i}(R)|$, is simple, the information is produced by the pre-processing phase which keeps track of how many rows in the table matrix are used.

The other kind of set is of the type $\pi_{u_i}(\sigma_{u_j=v_j}R)$ in which v_j is the vertex associated to u_j in the partial solution being evaluated. Since tables are built based on the QVO, is impossible to end in a situation in which u_j has not a mapped *data graph* vertex. Evaluating the sizes of this kind of set cannot be done precisely without scanning the columns and looking for the one actually used, which is quite an expansive task. Thus, it has been chosen to trade off the size accuracy in favor of faster execution. In particular, as size, it is used the number of edges present in the set $\sigma_{h(u_j)=h(v_j)}R$, which can be computed comparing two offsets inside the table matrix. Due to collision, however, this number can be bigger with respect to the actual hash set size.

After having evaluated all the correct tables, it is possible to communicate the details of the minimum set M to the next function.

Propose: read minimum set

Reading the minimum set means retrieving the hash values of the vertex ids, and is done differently based on the type of set. By convention R_M is the table containing the minimum set. In the case of an indexing set (a set from a table in which the current query vertex is the indexing vertex), it is possible to linearly scan the rows of the table matrix in R_M and stream out the indices, i.e the hash values, which are used. However, in the case of an indexed set, the above approach is not replicable on the columns, since columns have indices with smaller bit-width. The solution used is to read the edges in the set $\sigma_{h(u_j)=h(v_j)}R_M$, for each of them check if the indexing vertex is equal to v_j, thus avoiding collisions errors, and then computing the hash of the indexed vertex. In this way is possible to re-obtain the full-length set of hashes.

3.3.2 Intersect

The intersection is done by one monolithic function checking, for each element of the minimum set, if it is present in every other set. Given the current query vertex u_i, checking the existence of an element $a \in M$, which is already a hash value, in a set is done:

- For an indexed set $\pi_{u_i}(\sigma_{u_j=v_j}R)$, by testing the offset's MSB located in row $h(v_j)$ and column a, where v_j is the vertex associated to u_j in the partial solution being evaluated.
- For an indexing set $\pi_{u_i}(R)$, by checking the existance of the set $\pi_{u_i}(\sigma_{h(u_j)=a}R)$.

As already explained in section 3.2.1, it is tested the absence of an element from a set, and not the presence, accepting a portion of false positives. The explanation for this choice comes from the fact that, due to hash collision, checking if an element is present in a set would have required reading a portion of edges sharing the same hash of the element, and then comparing the vertex ids. The set of hashes that have passed the intersection test is referred to as M_{inter}

3.3.3 Extract

The extract phase translates the hash values in vertex ids, exploiting two overlapping functions. The operation is required to be able to verify the correctness of new partial embeddings generated in the next steps.

Extract: hash to vertex ids

The inputs to this function are the hash values that have passed the intersection phase. To recover the vertex ids from the hashes it is again used the minimum set

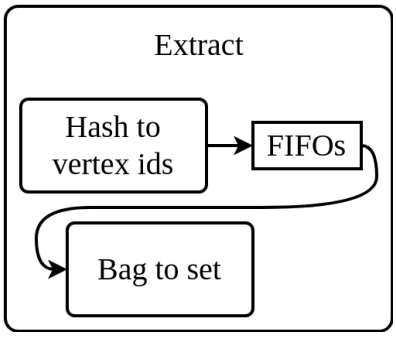

Figure 3.8: Extract pipeline.

M included in the table R_M, found in the *Propose* phase. Given the current query vertex u_i, the hash $a \in M_{\text{inter}}$, the main idea is:

- If M is an indexed set, look for the set of vertices $\pi_{u_i}(\sigma_{h(u_j)=h(v_j) \wedge h(u_i)=a} R_M)$ where v_j is the vertex associated to u_j in the partial solution being evaluated. In other words, taking only the indexed vertex ids from the edges compatible with $(h(v_j), a)$. In the ideal case of no hash collision this set has only one vertex inside, in reality, is a small bag (or multiset) with possibly repeated elements.

- If M is an indexing set, read the set of vertices $\pi_{u_i}(\sigma_{h(u_i)=a} R_M)$, which are the vertices from the edges having the indexing vertex hash equal to a. Again, in case of no hash collision, this set would be a bag composed of the same vertex repeated many times, and taking one of them would be enough.

The found bag contains the correct vertex ids but with some elements repeated more than once, which must be discarded since they would cause at the end duplicated solutions. In the end, every hash generates one bag of vertex ids.

Extract: bag to set

Passing from a bag to a set requires removing all the duplicates. This operation is done by an iterative process: every time a new element is read it is compared to the ones already present in the output set, if it is not part of it then it is added. Since this procedure is in worst-case quadratic in the number of elements of the bag ($\frac{n(n-1)}{2}$ comparisons in case of a bag of n distinct elements), it is better to evaluate smaller but numerous bags than having a single very large multiset.

3.3.4 Verify

Approximate intersection and hash collision make mandatory a verification step before generating new partial solutions. This final step is accomplished by three pipelined functions.

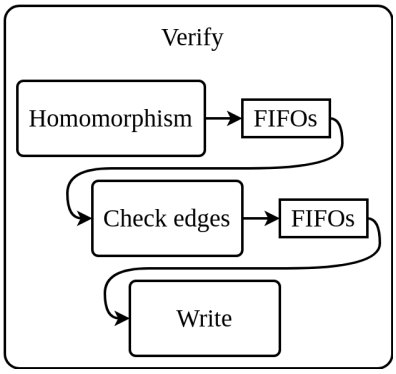

Figure 3.9: Verify pipeline.

Verify: homomorphism

Subgraph homomorphism has the same definition as the one given for subgraph isomorphism in section 1.1.1, with the only difference that instead of searching an injective function, it is searched a mapping between query vertices and data ones. This discrepancy implies that a *data graph* vertex can appear more than once in a solution. Join-based approaches intrinsically enumerate homomorphism, requiring an additional step to evaluate isomorphism which discards solutions with duplicated vertices. The purpose of this function is to verify that the new proposed candidates for the current query vertex are not already in the partial solution being currently evaluated.

Verify: check edges

Since the intersection is done with hashes following an approximative method, it is not guaranteed that all the vertex ids in input to the function are valid mapping for the query vertex being evaluated. Given the current query vertex u_i, verifying the correctness of the candidate v_c for u_i requires checking for each edge of u_i if exists also for v_c. Actually, it is enough to consider only the edges that connect u_i to already evaluated query vertices, otherwise, every edge would be verified twice. As an example, to check if v_c complies with the edge between u_i and u_j), with u_j mapped to v_j, it is possible to look in the set $\sigma_{h(u_j)=h(v_j) \wedge h(u_i)=h(v_c)}R$ and search for the edge composed by v_c and v_j.

Verify: write

The last function of the pipeline is in charge of appending the mappings for the current query vertex to the partial solution, writing it to the partial solution FIFO or to the output FIFO if every query vertex has been evaluated. This function is also in charge of keeping track of the number of partial results present in the FIFO. It is crucial for the correctness of the kernel to know how many partial results are still circulating in the multiway join, in order to avoid misbehavior in case of zero matching. This function is the only one that removes and produces new partial solutions: in fact, in case of an empty set intersection, the void solution still proceeds through the pipeline until is dismissed here.

Motivation to design choices

Another possible solution, considered in first place, was doing a normal set intersection, reading the minimum set vertices from memory and probing each of them against the other tables, spending time to avoid collision errors. This solution would have not required the *Verify phase* nor the bag to set in the *Extract* phase. However: (1) reading the minimum set from memory, due to collisions, still requires translating it from a bag into a set, moving the operation only earlier in the pipeline, and (2) avoiding collision error in the intersection requires reading the actual edges and comparing them to the vertex id probed. In the current implementation, the latter point is still done on the Verify phase but only on the hashes that have passed the intersection, saving some access in memory.

3.4 Testbench

The role of the testbench is to verify the correctness of the kernel by comparing the result to the one produced by a golden model. The testbench is accepting five files in input:

- *queryOrder.txt*: contains the QVO.
- *queryLabels.txt*: contains the labels associated at each query node.
- *queryEdges.txt*: contains the edges of the *query graph*.
- *dataLabels.txt*: contains the labels associated at each data node.
- *dataEdges.txt*: contains the edges of the *data graph*.

Labels and edges have been separated into two files only for test purposes since it was easier to modify the cardinality of the label set without having to rewrite completely the graphs.

First of all, the testbench is allocating the memory space in which the kernel will store all the data structures, this portion of memory is also initialized to all 0s. Then it reads the input files and streams to the kernel in order: the QVO, the query edges, and two times the data edges. After these operations, it waits for the results from the kernel. The embeddings are then counted and compared to the result generated by a golden model, which is the NetworkX implementation of subgraph isomorphism based on VF2, and exploration-based algorithm [13].

Chapter 4

Results

The design has been synthesized targeting the Ultra96 board powered by a Zynq UltraScale+ MPSoCs, reaching a maximum frequency of 215 MHZ. The table below reports the resource utilization.

Resource	Available	Used	Percentage
CLB LUTs	70560	40726	57.72%
CLB Registers	141020	68881	48.81%
CARRY8	8820	774	8.78%
F7 Muxes	35280	1447	4.10%
F8 Muxes	17640	209	1.18%
CLB	8820	8147	92.37%
LUT as Logic	70560	32639	46.26%
LUT as Memory	28800	8087	28.08%
Block RAM	216	126.5	58.56%

Table 4.1: Resource utilization

Setup

The performance of the design has been tested using part of real-world datasets, taken from reference [14]. The characteristic of the *data graph* used are reported in table 4.2. Since the original graphs are unlabelled, each node has been assigned a label chosen from a set of 4, arbitrarily small to not make the problem trivial. The set of subgraph queries used is shown in figure 4.1.

Name	$\|V(g)\|$	$\|E(g)\|$	$\|\Sigma\|$
Facebook5	1724	5000	4
Facebook8	1820	8234	4
Wikipedia10	1826	10000	4
Wikipedia23	3048	23689	4

Table 4.2: Dataset used

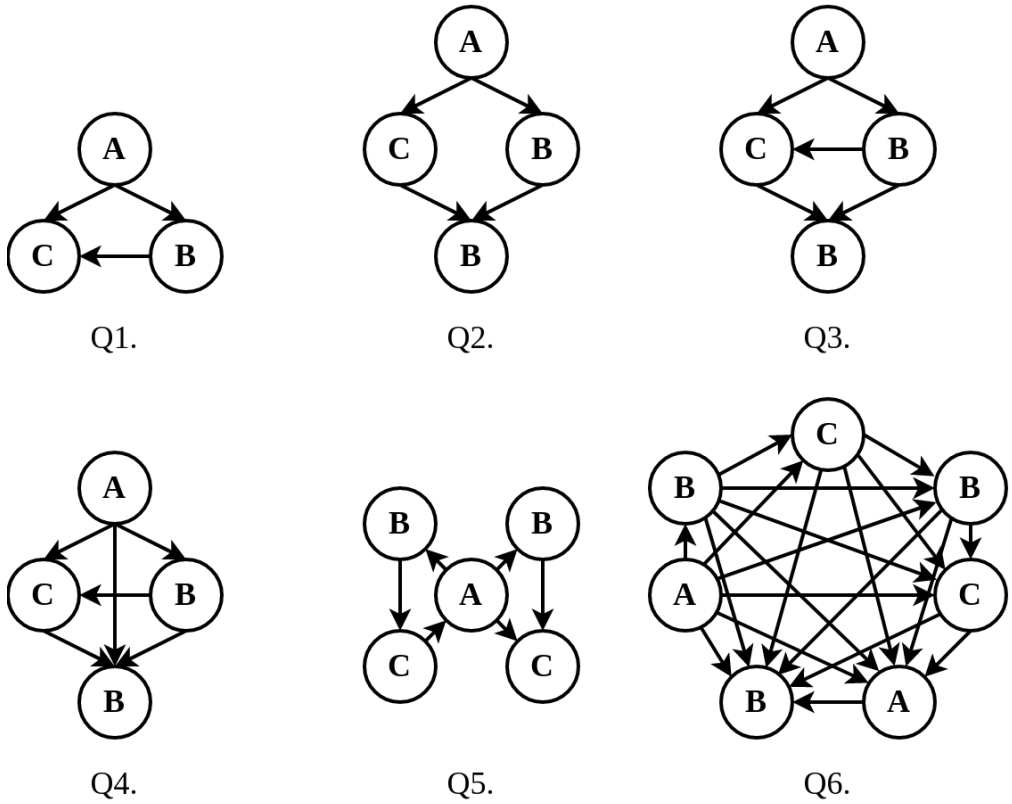

Figure 4.1: Subgraph queries.

Software and hardware comparison

The software implementation of subgraph matching from NetworkX is compared to the cosimulation of Vitis™ HLS, setting a latency for the memory equal to 64 clock cycles. The results shown in figure 4.2 are promising, suggesting the hardware solution as three orders of magnitude faster than the software one.

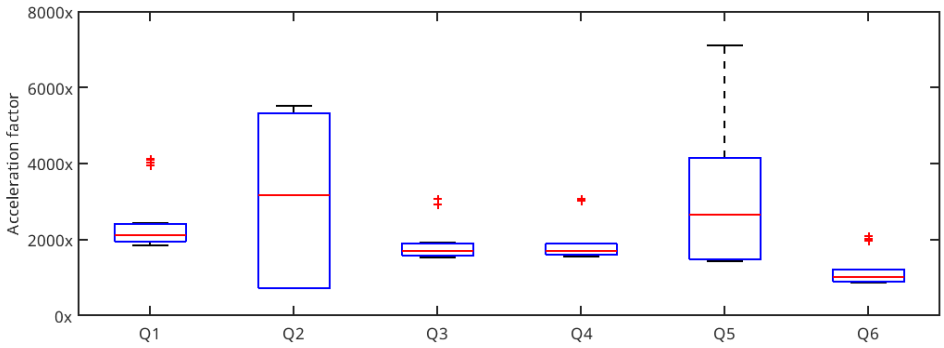

Figure 4.2: Boxplot representation of acceleration factors using six queries.

Effects of QVO

The importance of picking a good query vertex order has been already explained in chapter 1. As an example, figure 4.3 shows the time spent to evaluate query Q2 on the graph *Facebook5* using different QVOs. In this case, the choice of a bad order can lead to doubling the execution time needed to retrieve the result. In particular, the two worst QVOs are the ones in which the sequence of nodes is partially not connected, generating a cartesian product between candidates.

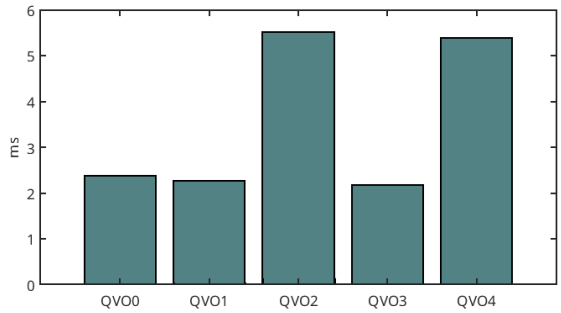

Figure 4.3: Query evaluation changing QVOs.

Chapter 5

Conclusion

5.1 Limitations

The main limitation of the current design is imposed by the possibility of the partial result FIFO to run out of space, deadlocking the kernel. In the current implementation, this FIFO has a fixed amount of space decided at compile-time that cannot be exceeded. Another current limitation is the QVO, which must be provided by the user. The kernel, right now, misses the heuristic logic to guess a good query vertex order.

5.2 Future work

There are several points that can be optimized in the design, as well as ideas that can be explored, some of which are:

1. Expanding all together the partial result by one query vertex at-a-time does not make sense, because implies saving all of them in memory. The right thing would be to take one partial result formed by one single vertex, extend it for all the query vertices, then output the result and starts again with another one. This approach would require near zero memory for the partial results FIFO. However, the described approach needs all the tables ready at the start. The approach currently used in the kernel needs instead only the tables related to the query vertex it is evaluating, building the base for a future version in which pre-processing and multiway join will work concurrently.
2. Designing an efficient FIFO able to save data on DDR when full will solves the limitation of the current design, described in the previous section.
3. Since several partial results share the same radix, finding a way to compress them could be an idea to reduce the partial result FIFO dimension.

4. Inserting a cache between the kernel and the main memory could significantly speed up the process. The spatial and temporal locality of the accesses in memory should be studied in future work.

5. Every partial solution is independent from the other ones, thus they can be expanded in parallel. This idea could be exploited by inserting more than one pipeline computing concurrently. If in theory work, the actual feasibility in hardware should be carefully studied.

Appendix A

Triangle query output size bound

Given a query $Q(a,b,c) = R(a,b), S(b,c), T(a,c)$, it is defined a new function

$$\mathbb{1}_{R(a,b)} \begin{cases} 1 & \text{if}(a,b) \in R, \\ 0 & \text{if}(a,b) \notin R, \end{cases}$$

Then it is possible to write the size of the query as:

$$\begin{aligned}
|Q| &= \sum_a \sum_b \sum_c \mathbb{1}_{R(a,b)} \mathbb{1}_{S(b,c)} \mathbb{1}_{T(a,c)} \\
&= \sum_a \sum_b \mathbb{1}_{R(a,b)} \sum_c \mathbb{1}_{S(b,c)} \mathbb{1}_{T(a,c)} \\
&\leq \sum_a \sum_b \mathbb{1}_{R(a,b)} \sqrt{\sum_c \mathbb{1}_{S(b,c)}} \sqrt{\sum_c \mathbb{1}_{T(a,c)}} \\
&= \sum_a \sum_b \mathbb{1}_{R(a,b)} \sqrt{|\sigma_{B=b}S|} \sqrt{|\sigma_{A=a}T|} \\
&= \sum_a \sqrt{|\sigma_{A=a}T|} \sum_b \mathbb{1}_{R(a,b)} \sqrt{|\sigma_{B=b}S|} \\
&\leq \sum_a \sqrt{|\sigma_{A=a}T|} \sqrt{\sum_b \mathbb{1}_{R(a,b)}} \sqrt{\sum_b |\sigma_{B=b}S|} \\
&= \sum_a \sqrt{|\sigma_{A=a}T|} \sqrt{|\sigma_{A=a}R|} \sqrt{|S|} \\
&= \sqrt{|S|} \sum_a \sqrt{|\sigma_{A=a}T|} \sqrt{|\sigma_{A=a}R|} \\
&\leq \sqrt{|S|} \sqrt{\sum_a |\sigma_{A=a}T|} \sqrt{\sum_a |\sigma_{A=a}R|} \\
&= \sqrt{|S|} \sqrt{|T|} \sqrt{|R|}
\end{aligned}$$

Where it is used three times the Cauchy-Schwarz inequality:

$$\sum_a x_a \cdot y_a \leq \sqrt{\sum_a x_a^2} \cdot \sqrt{\sum_a y_a^2}$$

It is also worth noticing that given a relation $R(a, b)$ of attributes A and B, under A fixed to a specific value a_1, then:

$$\sum_b \mathbb{1}_{R(a,b)} = |\sigma_{A=a_1} R|$$

Bibliography

[1] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. «Real-time twitter recommendation: Online motif detection in large dynamic graphs». In: *Proc. (VLDB)*. 2014, pp. 1379–1380 (cit. on p. 1).

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979 (cit. on p. 2).

[3] J. R. Ullmann. «An Algorithm for Subgraph Isomorphism». In: *Journal of the ACM* 23 (1976), pp. 31–42 (cit. on p. 2).

[4] Shixuan Sun and Qiong Luo. «In-Memory Subgraph Matching: An In-depth Study». In: *2020 ACM SIGMOD*. 2020, pp. 1083–1098 (cit. on pp. 2, 4).

[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. «Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows». In: *Proc. (VLDB)*. 2018 (cit. on p. 5).

[6] Todd L. Veldhuizen. *Leapfrog Triejoin: a worst-case optimal join algorithm*. 2012. DOI: 10.48550/ARXIV.1210.0481. URL: https://arxiv.org/abs/1210.0481 (cit. on p. 5).

[7] Hung Q. Ngo, Christopher Re, and Atri Rudra. *Skew Strikes Back: New Developments in the Theory of Join Algorithms*. 2013. DOI: 10.48550/ARXIV.1310.3314. URL: https://arxiv.org/abs/1310.3314 (cit. on pp. 6, 9, 10).

[8] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. «Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins». In: *ACM Trans. Database Syst.* 46 (May 2021), pp. 1–45 (cit. on p. 8).

[9] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. «Worst-case Optimal Join Algorithms». In: *PODS'12*. May 2012, pp. 37–48 (cit. on p. 8).

[10] Xilinx. *Vitis High-Level Synthesis User Guide*. 2021. URL: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/pragma-HLS-dataflow (cit. on p. 15).

[11] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, and You Peng. *FAST: FPGA-based Subgraph Matching on Massive Graphs*. 2021. DOI: 10.48550/ARXIV.2102.10768. URL: https://arxiv.org/abs/2102.10768 (cit. on p. 17).

[12] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. *Graph Processing on FPGAs: Taxonomy, Survey, Challenges*. 2019. DOI: 10.48550/ARXIV.1903.06697. URL: https://arxiv.org/abs/1903.06697 (cit. on p. 17).

[13] *NetworkX documentation*. 2022. URL: https://networkx.org/documentation/stable/reference/algorithms/isomorphism.vf2.html (cit. on p. 31).

[14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on p. 32).