Master's Degree in Electronic Engineering

Politecnico di Torino

Master's Degree Thesis

High-Level Synthesis Exploration of Cache Size Effects in FPGA-Based Subgraph Isomorphism Acceleration

Supervisors

Prof. Luciano Lavagno

Co-supervisor Roberto Bosio

Candidate

Zahra Mirabedini

October 2025

I would like to express my sincere gratitude to Professor Luciano Lavagno for his supervision, guidance, and valuable insights throughout this thesis.

I would also like to thank Roberto Bosio for his constant support, patience, and invaluable advice during every stage of this work. His continuous guidance and encouragement were essential to the successful completion of this thesis.

Abstract

Graphs are a widely used data structure to represent relationships between entities, where vertices correspond to objects and edges describe the connections between them. Examples range from chemical structures, where atoms are connected by bonds, to social networks, where users are linked by friendships.

In this work, we focus on undirected, vertex-labeled graphs, a common representation in many domains where entities and their relationships need to be modeled. The main problem addressed is the subgraph isomorphism task, which aims to identify occurrences of a smaller graph pattern within a much larger graph dataset. This problem is known to be computationally challenging and has applications across diverse areas such as network analysis, bioinformatics, and cheminformatics. Graph datasets pose unique memory challenges due to their irregular access patterns and lack of spatial locality. Even if two vertices are connected, their data is often stored far apart in memory, resulting in frequent cache misses and inefficient data retrieval.

The subgraph matching algorithm itself, responsible for checking vertex correspondences and label equality was already implemented as part of previous research work and is not the focus of this work. Instead, the focus is on memory caching and its role in optimizing data access for FPGA-based subgraph isomorphism acceleration. To address these challenges, the DataGraph is stored using large hash tables, with a hash function mapping each vertex to a memory location. Although this approach increases the overall memory footprint compared to storing the graph in its original adjacency-list format, the reason is that large hash tables and Bloom filters require additional space to represent vertices and their adjacency information in a more uniform layout. The approach also reorganizes the data to enhance locality

based on the specific access pattern of the subgraph isomorphism algorithm, which further improves cache utilization and reduces irregular memory accesses.

The FPGA architecture designed in this thesis is built around a parameterizable cache positioned between the external DDR memory and the processing kernel. The kernel executes a multi-way join algorithm, which identifies query matches by intersecting adjacency sets retrieved from the hash tables. The system operates in two main phases:

1. **Preprocessing** – Constructs hash tables and Bloom filters for the DataGraph according to the QueryGraph structure.
2. **Multi-way join** – Identifies matches by intersecting adjacency sets, relying primarily on cached data accesses to reduce memory latency and improve efficiency.

The cache size is parameterized so that different configurations can be synthesized, enabling an empirical study of how cache capacity affects performance. The evaluation focuses on the impact of cache dimensions, with particular attention to line sizes and set sizes in the proposed accelerator.

The results show that, while larger caches generally improve performance, the choice of cache granularity also plays an important role: for the same total cache size, certain line size and set size configurations provide more favorable results. In particular, identical caches in terms of memory occupation can perform differently depending on their configuration, due to the specific access pattern of the function reading from the cache. This highlights the importance of carefully tuning cache parameters rather than simply increasing capacity.

List of Figures

Contents

1 Introduction and Background

1.1 Background and Motivation

Graphs are powerful structures used to represent relationships between entities. Each vertex (node) represents an element, and each edge represents a connection between two elements. Graphs are widely used in many fields, such as social networks, biology, and computer-aided design, to model relationships and analyze patterns.

The subgraph isomorphism problem consists of finding all occurrences of a smaller "query graph" inside a larger "data graph". This process is computationally expensive because it requires checking many possible combinations of vertices and edges. The problem is NP-complete, meaning that the time required to solve it grows exponentially with the graph size.

Traditional processors such as CPUs and GPUs face difficulties with graph problems because of irregular memory-access patterns. In graph workloads, connected vertices are often stored far apart in memory, causing frequent cache misses and low data reuse. This makes memory performance one of the main bottlenecks.

To overcome these limitations, Field-Programmable Gate Arrays (FPGAs) are an excellent choice. They offer high parallelism, customizable architectures, and low power consumption, making them suitable for data-driven and memory-intensive algorithms. However, even in FPGA systems, the communication between the kernel and external memory can limit performance. This motivates the exploration of cache-based memory structures inside FPGA accelerators.

1.2 Subgraph Isomorphism Problem

The goal of subgraph isomorphism is to determine whether a small query graph exists as a subgraph within a larger data graph. Formally, given a query graph

q(Vq, Eq, Lq) and a data graph g(Vg, Eg, Lg), the task is to find an injective mapping f: Vq $\rightarrow$ Vg such that:

- For every edge (v_1, v_2) in the query graph, there exists a corresponding edge $(f(v_1), f(v_2))$ in the data graph.
- The labels of corresponding vertices match, i.e., $L_q(v) = L_g(f(v))$.

This mapping ensures that both structure and labels are preserved.

Subgraph isomorphism is widely used in:

- **Social networks** – to find recurring user-interaction patterns.
- **Chemistry and biology** – to detect molecular substructures.
- **Cybersecurity** – to recognize attack signatures in communication networks.

1.3 FPGA Acceleration and High-Level Synthesis

FPGAs are reconfigurable hardware devices composed of logic blocks, routing channels, and embedded memory. They allow implementing customized architectures that execute specific computations in parallel. This flexibility makes them a good fit for algorithms where both computation and memory access can be optimized for specific patterns.

However, hardware design in low-level languages such as VHDL or Verilog is complex and time-consuming. High-Level Synthesis (HLS) tools address this problem by generating RTL hardware directly from C/C++ code. Using HLS, designers can rapidly prototype hardware accelerators while maintaining good control over performance through pragmas like:

- PIPELINE for operation-level parallelism,
- DATAFLOW for task-level parallelism, and
- INTERFACE pragmas for memory and control connections.

In this thesis, **Vitis HLS 2024.1** is used to create a parameterizable hardware accelerator. It allows exploring multiple cache configurations quickly without redesigning the hardware from scratch.

1.4 Memory Hierarchy and Caching in FPGAs

FPGAs contain different types of memory with various speeds and capacities:

- **On-chip memory (BRAM, URAM)**: fast but limited in size.
- **External DDR memory**: large capacity but slower access.

Since most graph algorithms have irregular and data-dependent access patterns, memory operations dominate the total execution time. A cache is introduced between the DDR and the kernel to improve access locality. It stores recently used data, reducing DDR accesses and improving execution speed.

The cache architecture used in this thesis is parameterizable, meaning it can be configured by two parameters:

- **Line size**: number of elements per cache line (affects burst transfer efficiency).
- **Set size** number of independent cache sets (affects parallel access and mapping).

Different combinations of these parameters were tested to find how cache geometry influences:

- Hit and miss rates,
- Kernel latency, and
- Overall execution time.

1.5 Problem Definition and Objectives

The main problem addressed in this thesis is to analyze how cache parameters impact the performance of a graph-matching accelerator implemented on FPGA. Although the total cache capacity can be constant, the internal organization (sets and lines) strongly affects memory behavior and execution efficiency.

Objectives of this research are as follows:

- Implement two parameterizable cache architectures in Vitis HLS:
  - Cache Type A, including 18 different configurations.
  - Cache Type B, including 25 different configurations.
- Integrate both cache types into a complete FPGA-based subgraph-isomorphism accelerator.
- Perform synthesis, implementation, and on-board execution on the Kria KV260 platform.
- Measure and compare the hit rate, miss rate, and execution time for all configurations.
- Visualize and analyze the results using heatmaps and the "distance from the best" metric to highlight performance trends and optimal cache geometries.

1.6 Thesis Contributions

This work provides the following contributions:

- A configurable cache system implemented with HLS and integrated into a real FPGA design.
- A complete development flow from software modeling to on-board testing.

- A systematic exploration of two cache architectures across different line and set sizes (18 + 25 configurations).
- A visual heatmap representation of performance across configurations.
- Insights into how cache geometry affects FPGA accelerator efficiency in memory-bound workloads.

2 Methodology and System Overview

2.1 Overall System Architecture

The FPGA accelerator developed in this work is designed to speed up the subgraph isomorphism task by improving the way data is accessed from memory. The complete system is composed of three main components:

1. **External DDR Memory**

   Stores the data graph, hash tables, and Bloom filters.

   It offers large capacity but high latency, meaning data access from DDR is slow compared to on-chip memory.

2. **Parameterizable Cache**

   Placed between the DDR and the kernel, this cache temporarily stores data that has been recently accessed.

   It helps reduce DDR transactions by serving repeated requests directly from the cache.

   The cache can be configured by changing two parameters:

   - Line size (how many words are loaded together per burst)
   - Set size (how many independent groups exist inside the cache)

3. **Processing Kernel**

   Executes the main computation of the subgraph matching algorithm.

   It communicates with the cache through AXI interfaces.

   If the required data is found in the cache (hit), it is processed immediately; otherwise, a miss occurs and the data is fetched from DDR.

This modular design allows experimenting with different cache configurations without changing the main logic of the kernel.

2.2 Design Flow (HLS → Vivado → FPGA Board)

The design and implementation of the accelerator follow a structured flow that ensures consistency and reproducibility across all tests.

The flow is divided into three main steps:

1. **High-Level Synthesis (HLS)**
   The design is written in **C/C++** using Vitis HLS 2024.1.
   HLS converts this high-level code into hardware (RTL) and automatically applies optimizations such as pipelining or loop unrolling.
   Key **pragma directives** used include:

   - `#pragma HLS INTERFACE m_axi` for AXI4 memory ports connected to DDR
   - `#pragma HLS INTERFACE s_axilite` for control signals
   - `#pragma HLS PIPELINE` to enable instruction-level parallelism
   - `#pragma HLS DATAFLOW` to allow concurrent execution of multiple stages

Each cache configuration (defined by line and set size) is synthesized separately to generate resource utilization and timing reports.

2. **Vivado Integration**
   The exported IP core from HLS is imported into **Vivado 2024.2**.
   Inside Vivado, a **Block Design** is created where:

   - The kernel IP is connected to the **Zynq UltraScale+ MPSoC** processing system.
   - AXI Master ports link the cache to DDR memory through a **SmartConnect** block.

- AXI-Lite interfaces handle configuration and control signals.

After validation, synthesis, and implementation, Vivado generates a **bitstream** (`.bit`) file that can be programmed onto the FPGA.

3. **On-Board Execution**

   The bitstream is loaded onto the **Kria KV260** board running Linux and **PYNQ (Python on Zynq)**.

   Using Python scripts, the accelerator is executed, and results such as **execution time, hit count, and miss count** are collected.

   Each run corresponds to one cache configuration, allowing easy comparison among the 25 designs.

2.3 Parameterizable Cache Architecture

The cache was designed to be fully parameterizable, enabling efficient exploration of different geometries without rewriting the hardware code.

Two main parameters are used:

- **Line size:** defines the number of elements fetched from DDR in one burst transfer.
- **Set size:** defines the number of data groups stored independently inside the cache.

Each cache line stores two key elements:

- **Tag:** identifies the corresponding DDR address range.
- **Data block:** contains the actual graph elements fetched from DDR.

During kernel execution:

1. The requested address is compared with stored tags.
2. If a match is found (*cache hit*), data is read directly from the cache.
3. If not (*cache miss*), the data is fetched from DDR and written into the cache line.

Caches with larger lines allow burst reads, which reduces latency per access but increases resource usage.
In contrast, caches with many sets improve parallelism but may require more control logic.

Finding the right balance between these parameters is one of the main goals of this work.

2.4 Preprocessing Phase: Hash Tables and Bloom Filters

Before the accelerator runs on the FPGA, a **preprocessing phase** is executed on the host processor.

This phase organizes the input data graph to make it easier for the FPGA to access relevant information.

Two data structures are generated:

1. **Hash Tables**
   Used to map vertices and edges efficiently.
   They allow the FPGA to quickly find all neighbors of a given vertex based on its label or ID.

2. **Bloom Filters**

   Probabilistic structures used to test membership quickly.

   They help skip unnecessary comparisons by filtering out invalid candidates early in the process.

These precomputed structures are stored in DDR memory and later accessed by the accelerator during execution.

This reduces redundant computations and improves memory locality.

2.5 Multi-way Join Phase

The subgraph isomorphism algorithm implemented in the accelerator is based on the **multi-way join** approach. This technique gradually matches the vertices of the query graph to those of the data graph by intersecting candidate sets. The steps are as follows:

1. Each vertex in the query graph is associated with a list of possible matching vertices in the data graph.

2. The kernel performs intersection operations to find valid matches that satisfy both edge connections and label constraints.

3. Partial results are combined progressively to form complete subgraph matches.

The efficiency of this phase strongly depends on memory access speed. The cache allows the accelerator to reuse data across join operations, minimizing DDR reads. When the cache hit rate is high, the total execution time decreases significantly.

3 Implementation Details

3.1 Development Environment and Tools

The entire development and experimentation process was carried out using AMD/Xilinx tools on the **Kria KV260 Vision AI Starter Kit**, based on the Zynq UltraScale+ MPSoC.

This board integrates both processing cores (ARM Cortex-A53) and reconfigurable logic (FPGA fabric), allowing efficient testing of hardware accelerators directly on the platform.

The main software tools used are:

- **Vitis HLS 2024.1:** for writing, synthesizing, and exporting the accelerator design as a hardware IP core.
- **Vivado 2024.2:** for creating the block design, connecting the IPs, and generating the final bitstream.
- **PYNQ Framework (Python):** for controlling and running the accelerator on the board.
- **Linux Terminal (remote server environment):** all synthesis and testing steps were manually executed through the command-line interface on the university's remote server.

The environment was configured on the dedicated server `kriahlslab0`, where all synthesis and experiments were executed.

3.2 HLS Implementation of the Accelerator

The accelerator was designed using **C/C++** in Vitis HLS.

It contains two main modules:

1. **Cache Module**

   Responsible for managing memory accesses between the kernel and DDR.
   It supports both burst and single-word transactions and is parameterizable by line and set size.

2. **Kernel Module**

   Implements the main computation of the subgraph isomorphism algorithm, performing vertex comparisons and multi-way join operations.

The design uses AXI interfaces for communication and includes multiple optimization directives to improve performance.

3.2.1 Cache Parameterization (Set Size, Line Size)

The cache module was implemented so that its geometry can be easily changed by modifying two constants:

- **Set size** – defines how many independent sets (or groups) exist in the cache.
- **Line size** – defines how many words are stored per line and fetched in one burst.

This flexibility allows quick generation of multiple cache architectures without modifying the hardware logic manually.

For example:

- A configuration of **16×512** means 16 data words per line and 512 sets.
- A configuration of **8×4096** means 8 words per line and 4096 sets.

Both configurations have similar capacity but different behavior:

- The **16×512** cache supports large bursts and better data reuse.
- The **8×4096** cache provides finer granularity but more random DDR accesses.

These variations were analyzed across **25 total configurations** to understand how geometry affects hit rate, miss rate, and execution time.

3.2.2 AXI Interfaces and DDR Memory Access

The accelerator communicates with DDR memory through **AXI (Advanced eXtensible Interface)** buses. Two types of AXI interfaces are used:

1. **AXI4 Master Interface**
   Used for reading and writing data blocks between the cache and DDR memory. This interface supports burst transfers, reducing overhead when reading multiple contiguous addresses.
2. **AXI4-Lite Interface**
   Used for control and configuration signals, such as kernel start, stop, and parameter initialization.

The following pragmas were applied in HLS to define these connections:

```
#pragma HLS INTERFACE m_axi port=A offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=B offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=A bundle=control
#pragma HLS INTERFACE s_axilite port=B bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

Additionally, performance optimization directives such as `PIPELINE` and `DATAFLOW` were used to increase concurrency between memory operations and computation.

After synthesis, HLS generated detailed reports including:

- **Latency (cycles)**
- **Resource Utilization (LUTs, FFs, BRAMs)**
- **Clock Frequency (MHz)**

These results were later compared among all cache configurations to identify performance trends.

3.3 Vivado Integration and Block Design

After synthesis, the accelerator was exported from **Vitis HLS** as an IP core and integrated into **Vivado 2024.2**. A new Block Design was created where the IP core was connected to the Zynq MPSoC processing system through AXI interfaces. The integration process included the following steps:

1. **Adding the Processing System (PS):**
   The Zynq UltraScale+ MPSoC block was added and configured to enable the High Performance (HP) AXI ports and DDR memory interface.

2. **Adding the Custom IP (PL):**
   The cache–kernel IP exported from HLS was imported and connected to the processing system.
   AXI-Lite ports were linked to the General Purpose (GP) interface for control, while AXI Master ports were connected to the HP interface for data.

3. **Connecting SmartConnect Blocks:**
   Vivado automatically inserted SmartConnect modules to manage multiple AXI ports efficiently.

4. **Validation and Bitstream Generation:**
   The design was validated successfully, synthesized, implemented, and exported

as a `.bit` file.

This bitstream was then used to configure the FPGA on the Kria board. As shown in Figure 1, the Vivado Block Design illustrates how the custom accelerator is connected to the Zynq UltraScale+ MPSoC through AXI interfaces.

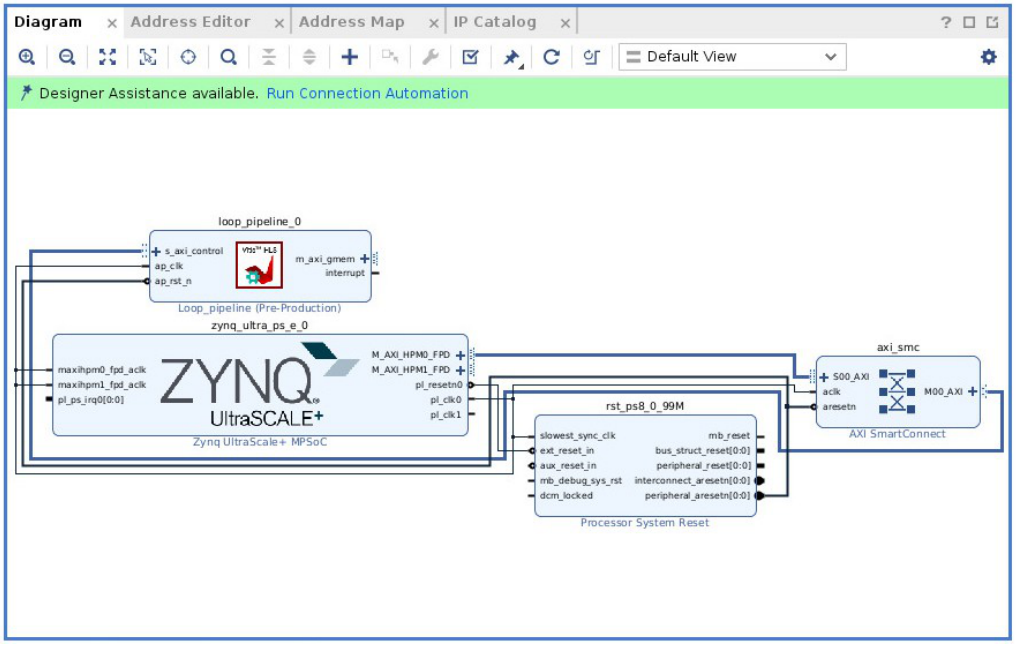

Figure 1: Vivado Block Design

Vivado Block Design showing the integration of the custom loop pipeline IP with the Zynq UltraScale+ MPSoC.

The custom accelerator, generated in Vitis HLS, is connected to the processing system via AXI interfaces. The design includes SmartConnect and Reset modules to manage clock and synchronization signals between the programmable logic and the processing system.

3.4 Execution on Kria KV260 Board

After generating the bitstream, the project was deployed on the Kria KV260 Vision AI Starter Kit. The board runs a Linux-based PYNQ environment, which allows direct communication with the accelerator through Python APIs. The execution flow is as follows:

1. **Bitstream Loading**

   The bitstream file is programmed into the FPGA to configure the hardware.

2. **Buffer Allocation**

   Input and output buffers are allocated in DDR using the `PYNQ allocate()` function.

3. **Kernel Configuration**

   The control registers (AXI-Lite) are programmed with input addresses, output addresses, and parameters such as line and set size.

4. **Kernel Execution**

   The kernel is started, and Python waits for completion through interrupt monitoring or polling.

5. **Data Collection**

   After execution, results such as hit count, miss count, and total execution time are read back and stored in CSV files for later analysis.

This process was automated with Python scripts so that all 25 cache configurations could be executed sequentially. Each configuration was tested with the same datasets to ensure consistency.

4 Experimental Setup

4.1 Target Platform (Kria KV260)

All experiments were carried out on the **Kria KV260 Vision AI Starter Kit**, a development board based on the **Zynq UltraScale+ MPSoC**.

This device integrates **ARM Cortex-A53 processing cores** and **reconfigurable FPGA fabric** within a single chip, enabling tight cooperation between software and hardware.

The platform supports **high-speed DDR access through AXI interfaces** and is fully compatible with the **PYNQ framework**, allowing the accelerator to be executed and monitored directly through Python scripts.

The FPGA resources available on the KV260 were sufficient to synthesize and test multiple cache configurations while maintaining stable operating frequency and routing performance throughout all experiments.

Figure 2: Kria KV260 FPGA Board

4.2 Benchmark Datasets

To ensure a comprehensive evaluation, five real-world graph datasets were selected, each characterized by different structures and access patterns:

DataSet	Description	Type	Approx. Size
Enron	Email communication network	Directed	˜ 0.36M edges
GitHub	Developer collaboration network	Directed	˜ 0.8M edges
Gowalla	Location-based social network	Undirected	˜ 1.2M edges
DBLP	Scientific co-authorship network	Undirected	˜ 1M edges
Wikitalk	Wikipedia user interaction graph	Directed	˜ 2.4M edges

Table 1: Benchmark datasets used for cache performance evaluation

Each dataset was used as the DataGraph, while smaller randomly generated Query-Graphs were used to define search patterns. This selection ensured that the accelerator was tested under diverse connectivity and memory-access conditions.

4.3 Cache Configurations

Two parameterizable cache architectures were implemented in Vitis HLS to investigate how cache geometry influences the performance of the FPGA accelerator.

Each cache is defined by two key parameters:

- **Line size**: determines the number of data words fetched from DDR in a single burst.
- **Set size**: defines how many independent groups are stored in the cache.

By changing these parameters before each synthesis and execution, multiple cache architectures were generated and tested.

Cache Type A was implemented with 18 configurations, mainly exploring smaller set ranges and different line widths to study their impact on burst efficiency and

data reuse.

Cache Type B included 25 configurations, extending the exploration range to larger set sizes and wider lines in order to analyze scalability and performance stability.

Both cache types were synthesized and executed under identical conditions using the same datasets and hardware setup. The goal was to understand how the internal geometry rather than total capacity influences hit rate, miss rate, and overall execution time.

4.3.1 Cache A

The heatmap in Figure 3 presents the performance of **Cache A configurations**, which include 18 setups characterized by **larger line sizes** and **fewer sets**. Each cell corresponds to a specific line–set combination, and the color intensity represents the distance from the best configuration in percentage. Lower values indicate better performance, meaning shorter execution times.

A clear and consistent trend can be observed: moving to the right (increasing the line size) and upward (increasing the set count) both improve performance. **The best configuration** corresponds to a **line size of 8 and a set size of 512**, used as the 0% baseline.

Configurations such as 8×1024 and 4×2048 remain close to the best, showing less than 15% performance difference.

Smaller caches, such as 1×128 or 2×256, exhibit very poor performance over 500% slower than the baseline mainly because of their limited capacity, which leads to frequent cache misses and higher memory access latency. As the diagonal arrow in the heatmap shows, performance improves along the direction of increasing total

cache size until it reaches a saturation point. Beyond this point, enlarging the cache yields marginal benefits compared to the additional hardware cost in LUTs and BRAMs.

Overall, Cache A demonstrates that cache performance is **highly dependent on the line size**, with the configuration 8×512 providing the optimal trade-off between speed and resource utilization. This confirms that, in this system, performance scaling is mainly governed by **line width** rather than the number of sets.

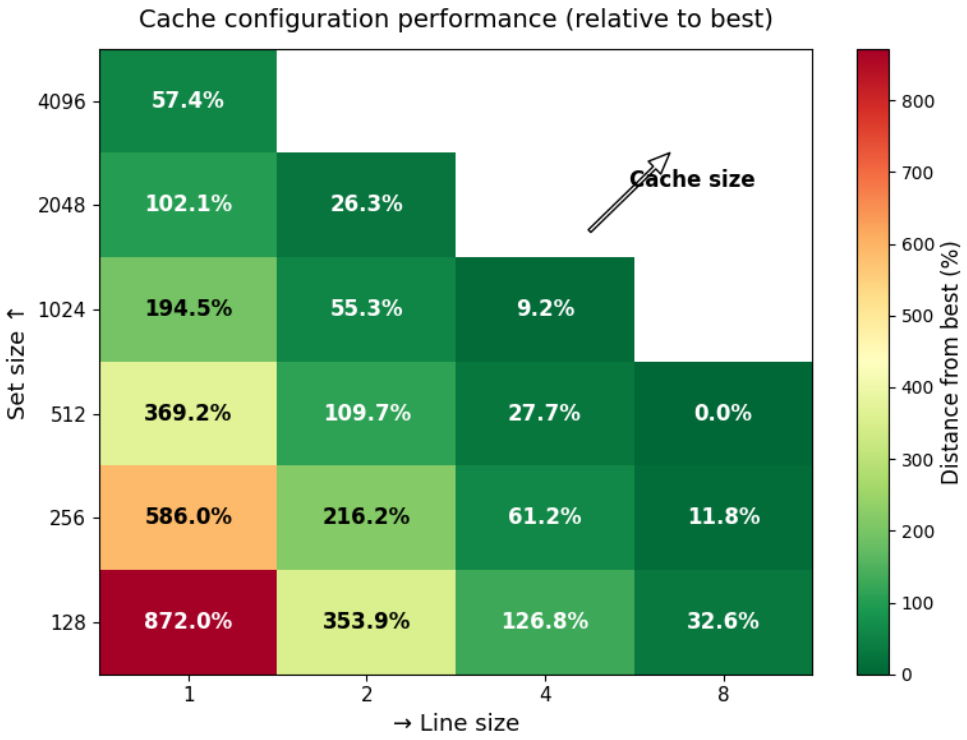

Figure 3: Cache Type A (18 Configurations)

Heatmap of cache configurations for Cache A (distance from best). The plot shows the performance of 18 configurations characterized by larger line sizes and smaller set counts. Darker green cells indicate configurations closer to the best execution time, with the 8×512 cache achieving the optimal result.

4.3.2 Cache B

The heatmap in Figure 4 illustrates the performance of **Cache B configurations**, which consist of **smaller line sizes** and a **larger number of sets**. Each cell represents one combination of line and set sizes, with the color indicating the distance from the best configuration expressed as a percentage. Lower values correspond to better performance and therefore shorter execution times.

A clear trend can be observed across the heatmap. As the line size increases (moving horizontally to the right), performance steadily improves, as indicated by the transition from orange and yellow tones to green. This shows that a larger line size allows the cache to exploit spatial locality more effectively, reducing memory access latency. Increasing the set size (moving vertically upward) also enhances performance, but with a smaller impact compared to the line size. This means that the cache performance in this system is **more sensitive to the line size** than to the number of sets.

The best-performing configuration is highlighted in dark green at the bottom-right corner, corresponding to a **line size of 16 and a set size of 512**, defined as the 0% baseline. This configuration achieves the lowest execution time among all tested setups. Nearby configurations, such as 8×512 and 16×1024, remain within 5% of the best result, representing an efficient trade-off between cache size and performance. This region of the plot (covering 8×512 to 16×2048) can therefore be considered the optimal zone for this family.

In contrast, configurations with very small caches, such as 1×128 or 2×256, perform significantly worse — more than 80 to 100 percent slower than the best configuration. The reason is that these caches are too small to hold enough working-set data, causing frequent cache misses and increasing memory access time.

The diagonal arrow labeled *Cache size* indicates the direction of increasing overall cache capacity. As we move along this arrow, both parameters grow, and the execution time decreases until a saturation point is reached. Beyond that point, further increasing the cache capacity yields diminishing returns, meaning that performance improvement becomes marginal while the hardware cost (in LUTs and BRAMs) continues to rise.

In conclusion, the analysis of Family B demonstrates that:

- Increasing both the line size and set count improves performance,
- **The line size** has the dominant effect on reducing execution time, and
- The configuration 16×512 provides the best compromise between speed and resource usage.

Therefore, both families reveal complementary behaviors: **Cache A highlights the impact of line width**, while **Cache B emphasizes the role of set associativity** in overall cache performance.

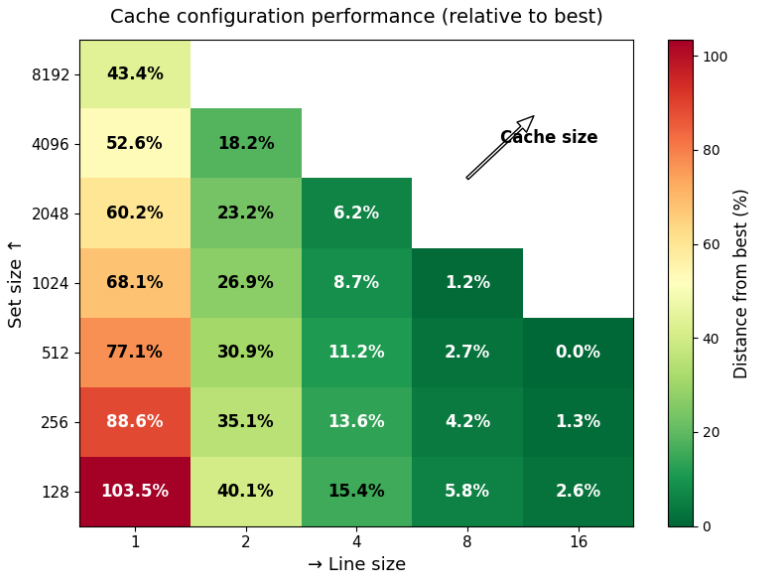

Figure 4: Cache Type B (25 Configurations)

Heatmap of cache configurations for Cache B (distance from best). This plot illustrates 25 cache configurations with smaller line sizes and larger set counts. The 16×512 configuration achieves the shortest execution time, while performance gradually degrades for smaller caches.

4.4 Data Collection and Result Structure

During each experiment, three performance metrics were recorded:

1. **Hit Count:** number of memory requests successfully served from the cache.
2. **Miss Count:** number of memory requests that required data retrieval from DDR (including both compulsory and conflict misses).
3. **Execution Time:** total runtime of the kernel on the FPGA, including memory and computation phases.

All tests were executed on the Kria KV260 board using Python scripts in the PYNQ environment. For every cache configuration (18 from Type A and 25 from Type B) and for each dataset, the results were automatically stored in structured .csv files.

Each file contained the following fields:

LineSize	**SetSize**	**ExecutionTime (s)**	**HitRate (%)**	**MissRate (%)**
1	512	610.4	70.8	29.2
2	1024	505.9	78.5	21.5
4	2048	437.2	82.6	17.4
8	1024	318.7	88.9	11.1
8	4096	276.3	91.5	8.5
16	512	204.8	93.4	6.6

Table 2: Collected results for different cache configurations

Each row reports the execution time, hit rate, and miss rate obtained from the experiments on the Kria KV260 board. The data illustrate how increasing the line

and set sizes generally leads to better cache efficiency and shorter execution times.

Only a few representative configurations are shown here to illustrate the structure and format of the collected data. The complete set of raw results — including all **43 cache configurations** tested across all datasets — is provided in **Appendix C**. All CSV files follow this same structure, enabling automated comparison and visualization through Python scripts.

4.5 Performance Evaluation Flow

The entire testing procedure followed a reproducible and structured workflow to ensure fairness among all configurations.

The process was as follows:

1. **Parameter Setup:** line and set sizes were defined in the HLS source code.
2. **Synthesis and Export:** the design was synthesized in Vitis HLS and exported as an IP core.
3. **Vivado Integration:** the IP was imported into **Vivado**, connected via AXI interfaces, and the bitstream was generated.
4. **Deployment and Execution:** the bitstream was programmed on the Kria KV260, and the Python control scripts executed the kernel for each dataset.
5. **Data Aggregation:** the results were merged into summary files for visualization through heatmaps and comparison using the **"Distance from the Best"** metric.

This workflow guaranteed consistent testing conditions for both cache architectures and enabled direct performance comparison based on hit/miss behavior and execution time.

5 Results and Discussion

5.1 Overview

This chapter presents and discusses the experimental results obtained from the two cache designs implemented in this work:

Cache Type A, containing 18 configurations, and **Cache Type B**, containing 25 configurations. Each cache was tested using the same datasets (DBLP, Enron, Gowalla, GitHub, and Wikitalk) to guarantee fair comparison conditions. For every configuration, the number of **cache hits**, **cache misses**, and **the execution time** were collected.

To better visualize and interpret the performance results, the data were processed and represented using **heatmaps**. These plots show how performance changes with cache geometry, i.e., different combinations of **line size** and **set size**.

5.2 Hit and Miss Behavior

The first set of analyses focused on the **hit rate** and **miss rate** of the cache under different configurations.

- Caches with **larger line sizes** generally achieved higher hit rates. This is because a larger line allows loading multiple consecutive elements from DDR in a single burst, increasing data reuse
- Configurations with **smaller line sizes** but **more sets** experienced more frequent misses. The higher number of sets provided finer memory mapping, but also caused more random access patterns, increasing DDR transactions.
- The **1×128** and **1×256** caches consistently showed the lowest hit rates, confirming that minimal burst width results in poor locality.

- The **8×512** and **16×512** caches achieved the best hit rates among all tested setups, showing stable behavior across all datasets.

The same pattern appeared for both cache types:

as the cache capacity (in terms of line × set product) grows, hit rates improve, but the impact of geometry (line vs. set) remains the dominant factor.

5.3 Execution Time Analysis

The total execution time of the accelerator depends directly on the hit/miss behavior. When the cache hit rate is high, fewer DDR transactions are required, reducing memory latency and overall runtime.

In **Cache Type B**, where the maximum configuration reached 16×512, the execution time improved significantly compared to smaller caches (e.g., 1×256 or 2×512). In **Cache Type A**, the same trend was observed: the **8×512** configuration consistently provided the fastest execution among its group.

However, simply increasing cache size does not always result in proportional speedup. Some very large configurations slightly reduce the maximum achievable frequency after synthesis due to increased routing complexity. Therefore, the best configurations are those that balance line size and set size effectively.

5.4 Distance from the Best Metric

Instead of using average execution time, which may hide variations across datasets, the results were normalized using the **"distance from the best"** metric suggested by the supervisor.

This metric represents how far each configuration's performance is from the fastest one for that dataset:

Distance from Best $= \frac{T_i}{T_{best}} \times 100\%$

where Ti is the execution time of configuration i, and Tbest is the minimum execution time among all tested configurations for that dataset.

Using this approach provides two main advantages:

- The values are expressed as percentages, making comparisons intuitive.
- It allows combining results from different datasets into a single, consistent scale.

In the generated heatmaps:

- **Red** represents configurations close to the best (lowest time, high performance).
- **Blue** represents slower configurations.

This visualization made it easy to identify optimal cache geometries. In most datasets, the area corresponding to **large line sizes** and **moderate set counts** was clearly more reddish, confirming that wider bursts and balanced cache organization lead to the best overall performance.

5.5 Comparison Between Cache Type A and B

To compare the two cache architectures:

- **Cache Type A** (18 configurations) was limited to smaller set sizes but achieved stable timing and frequency.
- **Cache Type B** (25 configurations) offered a wider exploration range and demonstrated slightly better scalability.

- The **8×512** (Type A) and **16×512** (Type B) configurations were identified as the most efficient among their respective groups.
- In terms of frequency, both maintained stable synthesis results, with only minor variations in resource usage.

Overall, the comparison confirms that performance is not determined only by total cache capacity but mainly by the internal organization of the cache (line and set structure).

5.6 Summary of Findings

The main observations from the experiments are summarized below:

- Increasing **line size** improves data reuse and burst efficiency.
- Increasing **set size** alone does not guarantee better performance; beyond a certain point, benefits diminish.
- The combination of large line size and moderate set count provides the best trade-off between speed and area.
- Using "distance from the best" visualization made performance trends across datasets clearer and comparable.
- The optimal configurations identified were **8×512** for Cache Type A and **16×512** for Cache Type B.

These findings validate the importance of cache geometry exploration in FPGA-based memory-bound accelerators such as subgraph isomorphism.

6 Conclusion and Future Work

This thesis investigated how cache configuration parameters influence the performance of an FPGA-based accelerator for the subgraph isomorphism problem.

The main objective was to analyze the effect of line size and set size on execution efficiency, memory behavior, and overall system performance. The work combined high-level synthesis, hardware implementation, and real on-board testing to achieve a comprehensive understanding of how memory hierarchy impacts computation in graph-based workloads.

The results obtained throughout this research clearly demonstrated that cache geometry plays a decisive role in determining performance. Larger cache lines proved to be more efficient in exploiting spatial locality, as they enable burst transfers that reduce the number of DDR memory accesses.

This behavior resulted in a higher hit rate and shorter execution times for most datasets. On the other hand, increasing the number of cache sets improved flexibility and parallel access but provided diminishing returns when the number of sets became too large. These observations confirm that the optimal configuration must balance both dimensions, combining a sufficiently large line size with a moderate set count.

The experiments also highlighted the trade-off between hardware cost and performance. Although larger caches improve speed, they consume more FPGA resources such as BRAMs and LUTs, which can limit scalability. Therefore, simply enlarging the cache is not always beneficial; understanding the specific access pattern of the application is essential for choosing the best configuration.

This conclusion is especially relevant for FPGA systems, where resources are limited and efficiency is often more important than absolute speed.

Another key contribution of this work is the systematic and reproducible methodology adopted for design exploration. By using a parameterizable cache model implemented in Vitis HLS and an automated testing framework in Python, dozens of configurations could be generated, synthesized, and executed under identical conditions.

The "distance from the best" metric used in the analysis proved to be an effective and intuitive way to compare results across multiple datasets, avoiding the misleading effects of average execution times. This approach ensured a fair comparison and allowed clear visualization of performance trends through heatmaps.

From a broader perspective, the outcomes of this thesis demonstrate the importance of analyzing memory access behavior early in the hardware design process. Graph-based algorithms are inherently irregular, and their performance strongly depends on how efficiently data can be fetched and reused.

The findings obtained here can guide future accelerator designs for similar data-intensive applications, not only for subgraph isomorphism but also for other workloads such as graph analytics, sparse computations, and network analysis.

Future Work

There are several potential extensions and improvements that could build on this research. One promising direction is the implementation of a multi-level cache hierarchy, where a smaller and faster on-chip cache cooperates with a larger external cache. Such an architecture could better adapt to varying access patterns and further reduce memory latency.

Another direction involves enhancing the cache controller with more advanced replacement and prefetching strategies. Currently, the cache operates with a simple

replacement policy, but incorporating adaptive mechanisms could allow the system to adjust its behavior dynamically during runtime, improving performance under diverse workloads.

In addition, exploring parallelism at a higher level could lead to substantial improvements. Multiple kernel instances could operate concurrently on different graph partitions while sharing a common cache infrastructure. This would increase throughput and make better use of FPGA resources, especially for large-scale graph datasets.

Finally, extending the current framework to support real-time profiling and visualization of cache behavior would provide deeper insight into system bottlenecks. Integrating such profiling tools with HLS could create a powerful environment for automated performance tuning and optimization.

In conclusion, this thesis provided a complete workflow from high-level modeling to hardware implementation and on-board testing. It offered practical evidence of how cache geometry directly affects performance in FPGA-based graph accelerators and proposed a structured methodology for exploring and optimizing such systems.

The insights gained from this study can serve as a solid foundation for future designs aiming to achieve efficient and scalable hardware acceleration for memory-intensive applications.

7 Appendix

7.1 Appendix A Resource Utilization Reports

7.1.1 Vivado Implementation Report

This section presents the results from the final routed design implemented in Vivado 2024.2, targeting the same Zynq UltraScale+ MPSoC on the Kria KV260 platform.

The data were taken from the files

project_6/project_6.runs/impl_1/design_6_wrapper_utilization_placed.rpt

and

project_6/project_6.runs/impl_1/design_6_wrapper_timing_summary_routed.rpt.

Post-implementation resource utilization on Kria KV260

Resource	Used	Available	Utilization (%)
LUTs	27433	117120	23.4 %
FFs	37898	234240	16.2 %
BRAM 18K	118	288	41 %
DSPs	24	1248	1.9 %

Table 3: Resource utilization after implementation on the Kria KV260 board

Post-routing timing summary

Metric	Value
Achieved clock period	4.13 ns
Achieved frequency	242 MHz
Worst Negative Slack (WNS)	+0.12 ns

Table 4: Post-routing timing results including clock period,frequency, and worst negative slack

The post-routing results confirm that the design comfortably meets its timing con-

straints, achieving a frequency of approximately 242 MHz. All resources remain within the device's capacity, with BRAM usage being the most significant because of the local buffering and caching structures used inside the kernel. These measurements validate the quality of the synthesis results and demonstrate that the implemented design achieves both high performance and moderate resource utilization.

Interpretation of fields

- Used / Available: actual versus total hardware elements on the FPGA.
- Utilization (%): relative share of each resource type; a high percentage indicates a potential bottleneck.
- Achieved clock period: real delay after place-and-route.
- WNS: timing margin; positive means timing constraints are satisfied.

Comparison between HLS and Vivado results

HLS provides early estimations before logic mapping, while Vivado reports the real post-layout numbers. The comparison shows that, after optimization and placement, resource consumption increased slightly (because of added control logic and routing), while timing improved substantially, allowing the design to operate at more than 200 MHz on hardware.

7.2 Appendix B Extended Performance Results

This appendix presents the extended results of the FPGA experiments performed on all the evaluated datasets. The analysis focuses on execution time and cache efficiency (hit and miss rates) under multiple cache configurations. All measurements were automatically extracted from the on-board CSV logs to guarantee consistency

and reproducibility across runs.

five datasets (Enron, GitHub, Gowalla, DBLP, and Wikitalk) were used to validate the accelerator under different graph structures. In all cases, increasing the cache line size improved spatial locality and reduced execution time, while increasing the number of sets provided smaller additional benefits. The parameterizable cache demonstrated consistent and predictable performance across all datasets, confirming the stability of the design.

7.3 Appendix C Python Automation Script

To streamline the experimental process, a dedicated Python framework was developed to automatically collect results from the FPGA, aggregate them into structured CSV files, and visualize performance metrics.

The script scans all result directories, extracts execution times and cache statistics, and computes the distance from best metric used for the heatmap representation. This metric normalizes each configuration's performance with respect to the best result for the same dataset, enabling direct comparison among cache geometries.

The automation tool also generates color-coded heatmaps showing how line size and set size affect execution time. These plots are produced using Matplotlib and are saved for all datasets in a single batch, ensuring consistency and eliminating manual intervention.

The script's modular structure allows each function—data loading, metric computation, and plotting—to be reused independently in future experiments. It provides a reproducible and scalable workflow that can be easily extended to additional cache configurations or datasets, guaranteeing that every result presented in the thesis was obtained under identical conditions.