POLITECNICO DI TORINO

Corso di Laurea Magistreale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Subgraph Isomorphism: parallelizing dataflow of an FPGA based architecture

Relatori
prof. Luciano Lavagno
co-relatore Roberto Bosio

Candidato
Giuseppe Salvatore Piazza

Academic Year 2023-2024

Abstract

Subgraph isomorphism is a well-known NP-hard problem involving a large graph, referred to as the **datagraph**, and a smaller graph, known as the **querygraph**. In this context, both graphs have labeled vertices, and an isomorphism exists when one or more subgraphs of the datagraph contain nodes with labels that match those of the query graph's nodes, and these nodes are connected by the same edges.

Currently, numerous CPU-based algorithms are available for this matching process. To enhance speed, GPU-based algorithms can be considered; however, they demand significant power, especially when high-bandwidth communication channels are involved. To achieve lower power consumption while leveraging highly parallel architectures, an FPGA-based core may be a more effective solution.

In this thesis, the LESS architecture, developed using Vitis HLS for Xilinx Kria boards, is optimized through parallelization to enhance performance. LESS implements the Multi Way Join (MWJ) algorithm, utilizing a hash table mechanism that is built starting from the query graph and enables efficient storage of data graph edges, thereby streamlining edge reading tasks. However, a significant bottleneck arises from DRAM access, as reading edges necessitates numerous read operations. The aim of this parallelization effort is to duplicate blocks that frequently access edge tables and distribute multiple instances of these blocks across various memory banks, ultimately achieving increased throughput.

To enable parallelization by a factor of N, the preprocessing phase has been adjusted to account for the increased number of tables that need to be populated, which is now multiplied by N. This necessitates early identification of the destination processing branch for each edge in the data graph. In the experiment conducted, second-degree parallelism was explored, utilizing the most significant bit (MSB) of the node's hash to designate the destination memory bank for each edge. The remaining bits of the hash continue to be employed as in previous implementation.

Instances of the MWJ algorithm can operate independently, even when tasks are duplicated, because each instance is linked to a separate memory bank. Once each set of edges has been processed by its belonging branch, all data can be converged into single stream by an additional task which must mix streams in correct way.

Another factor to consider during duplication is the format of streamed data across channels between tasks. Nearly half of the work involves decompressing certain streams (originally compressed to minimize streaming effort) and inserting extra tuples into specific streams to ensure data alignment across different channels.

Contents

Chapter 1

Background

1.1 Graph Isomorphism

In the field of graph theory, there is an isomorphism between two graphs **A** and **B** if exists a bijective function which maps all vertex of A to all ones of B:

$$f : V(A) \rightarrow V(B) \textit{ where V() is intended as set vertex of a graph}$$

and, to preserve edges integrity (edge-preserving bijection), it's required that, given two nodes **u** and **v** of graph A that are adjacent, the mapped nodes **f(u)** and **f(v)** of graph B must also be adjacent, for every node of sets V().

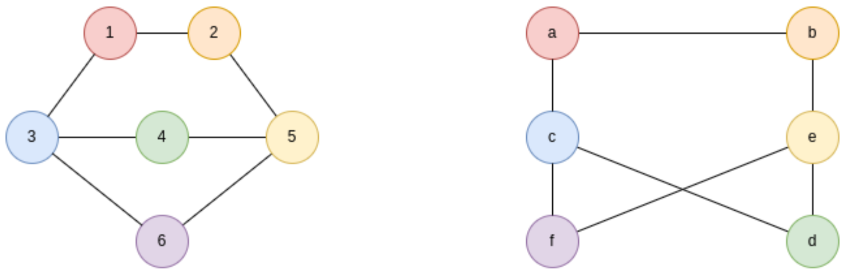


Figure 1.1. An example of isomorphism of two 6 vertex non-labeled graphs: for simplicity the map function is the cardinality of letters: (1,2,3,4,5,6) to (a,b,c,d,e,f)

It's evident, given also the figure above, that this relation is useful to determine equivalence between graphs, especially when these are graphically represented in totally different ways.
As stated before, the give definition is limited to unlabeled graphs, additionally it need to be updated if other characteristic of edges (directionality and weight) are considered.

1.1.1 Labeled graphs isomorphism

The definition previously given can be extended to support labeled graph by adding the "label-preserving" requirement, so the updated definition will be:

$\forall (u, v) \in V(A)$, calling f(u) and f(v) as w and x respectively, A and B are isomorphous if $\forall (w, x) \in V(B)$ L(u)=L(w) and L(v)=L(x), where L() is the label of a node.

Following figure shows the updated version of previous example, labels are added in form of node's colors (red, green, blue) while (1,2,3,4,5,6) to (a,b,c,d,e,f) mapping is kept:

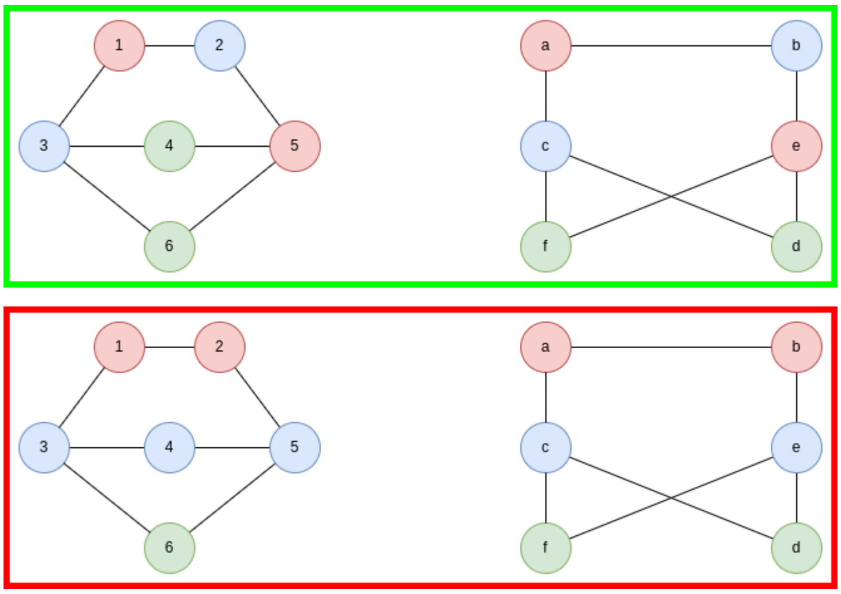


Figure 1.2. Isomorphism of two 6 vertex labeled graphs

The examples inside the green rectangle are isomorphous other ones inside red rectangle are not. Following a table detailing comparison between vertices in the second case:
In last example, the requirement during mapping from node 4 of graph A to node d graph B isn't satisfied since the first node has blue label, second one green.

1.2 Subgraph Isomorphism

The concept of graph isomorphism can be modified to define subgraph isomorphism (a query operation modeled with graphs): practically it consist in finding all the subgraphs of a bigger graph, called **Datagraph** because is the actual database, which can play the role of graph A, the role of graph B is assumed by the **Querygraph** which is the query, answering to "how many istances of Querygraph there are in the Datagraph?". This problem is NP-hard. [2]

This task is fundamental in many today application, one example can be given in the field of social networks: assuming the Datagraph contains users and groups: each node has an ID which is unique, and its label describes the type of node (if it's an user or a group), to build an algorithm to perform an action like suggesting a friendship between two users that both belong to a group, all possible relations to process can be proposed by a subgraph isomorphism task which involve a querygraph having a node with a label identifying the type "group" linked to another two nodes having a label identifying the type "user", the task will find how many "relations" exist in the social network. [5]

Following an example using a small datagraph and an Y shaped querygraph, this example can be seen as searching 4 chemicals elements connected with the pattern dictated by querygraph inside a molecule modeled by datagraph.

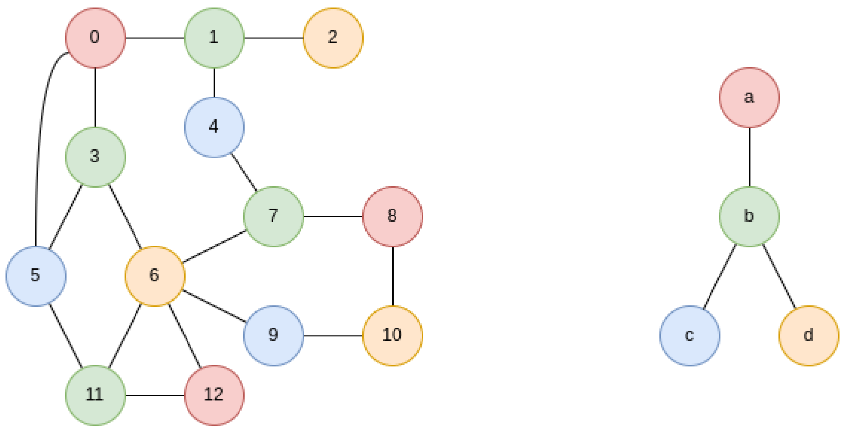


Figure 1.3. Submorphism using datagraph with 13 vertex and 17 edges

In the example above, colors are again used as labels and there are 4 matches to query (a,b,c,d) involving these sets of nodes of datagraph: (0,1,4,2), (0,3,5,6), (8,7,4,6), (12,11,5,6).

1.3 FPGA Basics

To exploit high parallelism architectures for specific tasks custom circuits need to be built, earlier solutions relied on ASICs (application specific integrated circuit) which are described using HDLs (Hardware Description Languages), then they're mapped to a specific technology and, then, actual circuit is realized by a chip foundry.

Relying on ASICs to realize custom architecture can benefit from:

- High clock speeds, since each logic block can be optimized in terms of delays.
- Achieving low power consumption thanks to synthesizer algorithms.

but, going for this method involves some challenges:

- High costs in terms of testing (RTL simulation, second delay aware simulation, drastic consequences in case a bug is discovered after actual circuit realization).
- Difficulty in terms of updating the architecture after realization.

The solution which merge flexibility of typical software algorithms and the customization of the hardware is represented by FPGAs (Field Programmable Gate Arrays) that, as name suggests, are particular architectures formed by arrays of elementary blocks, typically Flip-Flops and LUTs (LookUp Tables), which can be programmed by the final user to act as a specific gate-based logic circuits. Following typical block of an FPGA's element:

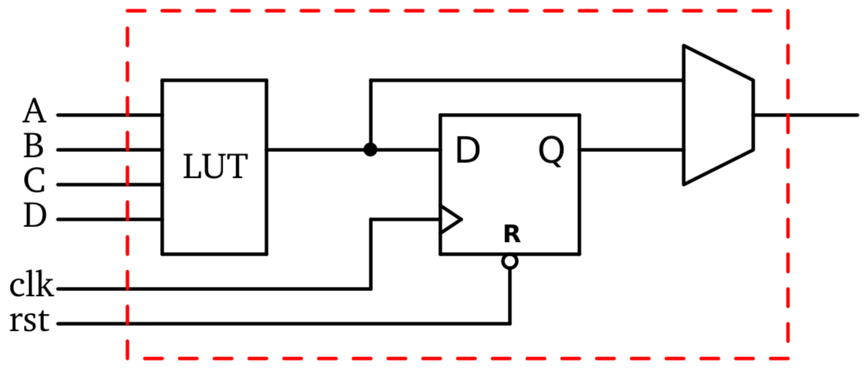


Figure 1.4. CLB (Configurable Logic Block) of an FPGA

In the figure above the representation of a CLB give an idea that this block can be used to described any 4 input to 1 output combinatory network, just load its related truth table inside the LUT memory and set the mux to bypass the flip flop element.

Modern FPGAs includes thousands of these blocks and, additionally, to avoid a large wast of CLBs, other block like adders,multipliers and BRAMs (block-RAMs, usefull to avoid wast of CLB's flip-flops) are included. FPGA itself has only elementary blocks and volatile memories, thus at startup a specific procedure is needed to setup its clock source, load the bitstream (the binary code which fill LUTs and other configuration registers of FPGA's elements) and monitoring its correct execution.

1.3.1 FPGA boards and SoCs

To assist the FPGA during startup additional hardware is needed, like PLLs (phase-locked loop) to generate clock at which CLBs will run, at least one EEPROM (Electrically Erasable Programmable Read Only Memory) to store the bitstream (thus making possible to shutdown the board without losing the logical configuration of the circuit), a MCU (MicroController Unit) containing a firmware capable to load bistream from EEPROM and upload it to FPGAs configuration memory blocks and, finally, voltage regulators to provide power to all previously mentioned hardware.

An FPGA board provides more functionalities than the minimum ones mentioned before: typically a UART/USB adapter is required to load the bitstream to FPGA's volatile registers or EEPROM, in the education purpose boards some push-buttons, discrete LEDs and displys are also included, modern boards also provide an ARM CPU instead of a simple MCU to give the possibility to create an architecture where software based code can cooperate with FPGA as it is an HW-accelerator.

Nowadays advanced FPGA boards are equipped with a SoC (System-On-Chip), an unique chip which includes fpga, the CPU and basic pheripherals. The board remaining space can host RAM banks (up to some GBs), high bandwidth communication channels thus making it possible to develop more advanced architecture built by an FPGA controlled by CPU running an embedded operating system.

Xilinx Kria & Zynq UltraScale+

A practical example of the already described type of FPGA boards is represented by AMD Xilinx Kria K26 (LESS project, which is used to carry out this experiment, is tested on this board), that mounts a Zynq UltraScale+ SoC.
This chip (shown in figure below) includes different processing elements (APU,

RPU...), the FPGA, controllers (USB, I2C...), DDR interface and Master AXI (used in Vitis HLS project to connect memories to FPGA) controllers.

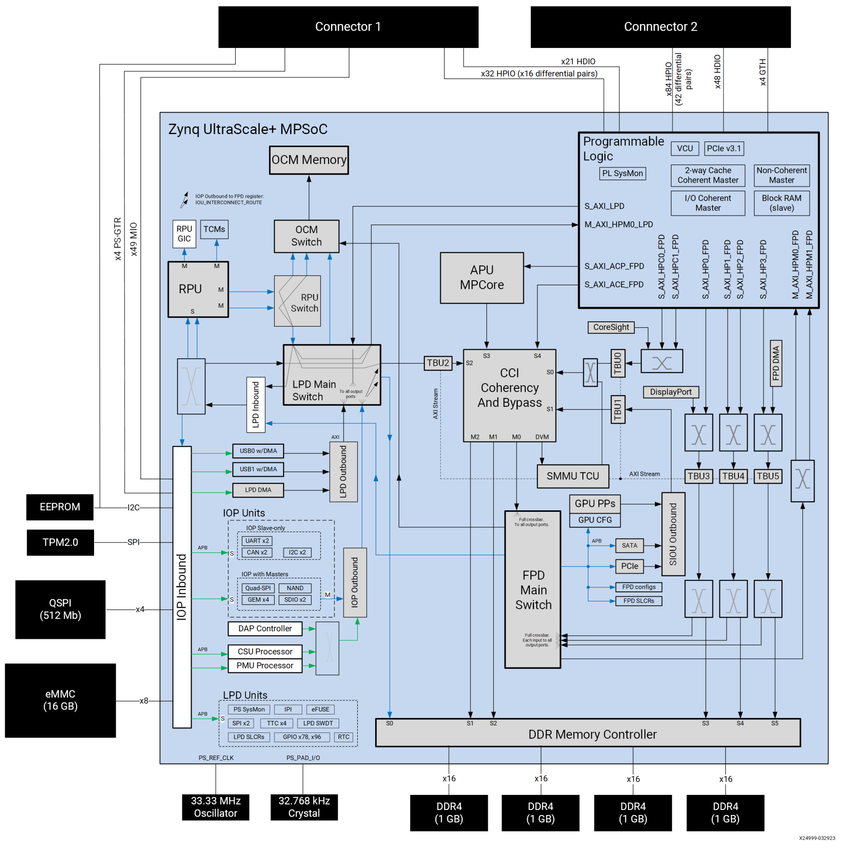
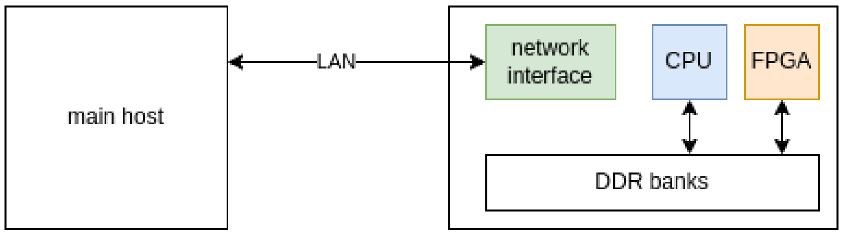


Figure 1.5. Source: AMD Kria K26 SOM Data Sheet (DS987)

1.3.2 OS cooperation

More advanced architectures like ones just described, thanks to the presence of an operating system, permit to develop projects which require advanced functionalities.
The study case of this thesis, subgraph isomorphism, can exploit a Linux-based operating system to benefit of network stack (which is easily managed by an OS) and high levels programming languages like Python (to assist the execution of the FPGA core feeding it with the data coming from network, transferred, for example, with SSH) to build a datacenter infrastructure as shown in the figure below:



1.3.3 FPGA in datacenters

The illustration above give the idea of a very common situation: "main host" block refers to the computer which hosts the server (backend, frontend or both) thus meaning that it's running the web application (for example a social network), when a runtime need to perform a graph-based operation, it converts, if needed, both database and query in a graph-based format and transfer it to the FPGA trough "LAN" link, which in the schematics is represented in a trivial simplification of local network. The CPU of the FPGA runs an overlay (it may be built in python or in C++ depending from available and actually used platforms and the workflow for the specific FPGA board that is being used) playing the role to load the database (and the query) in a portion of DDR, pass the pointer the arbiter of memory channels where FPGA is connected to, and, vice-versa, wait stops signals to read data to pass back to local network.

The idea can be extended to a larger structure: a cluster of servers, the balancer which not only manages the request servicing between servers but also decides to assign the most desirable choice in terms of which FPGA board should solve a specific query.

1.4 High level synthesis

Designing an architecture which fits in FPGA is typically carried out using HDLs, the strenght point of these languages is that the architecture can benefit from accurate description of hardware blocks if the designer has considerable digital design experience, additionally, when designing a block, it must be thought in terms of resulting hardware, even more modern languages (like SystemVerilog) offer behavioral descriptions methods.

HLS (High Level Synthesis) is a newer paradigm that permits to design an architecture by describing it trough an "algorithmic" as it was a software (in fact it's also called **algorithmic synthesis**), thanks to this new paradigm its possible for designers to focus more on what an architecture is going to do rather than how it will be made.

In the years different HLS languages and tools have been developed to accomplish synthesis using this new methodology, some are academic, other ones are commercial. In this thesis Vitis HLS, developed by AMD for Xilinx boards, is used which is based on C++. [6]

1.4.1 Vitis HLS

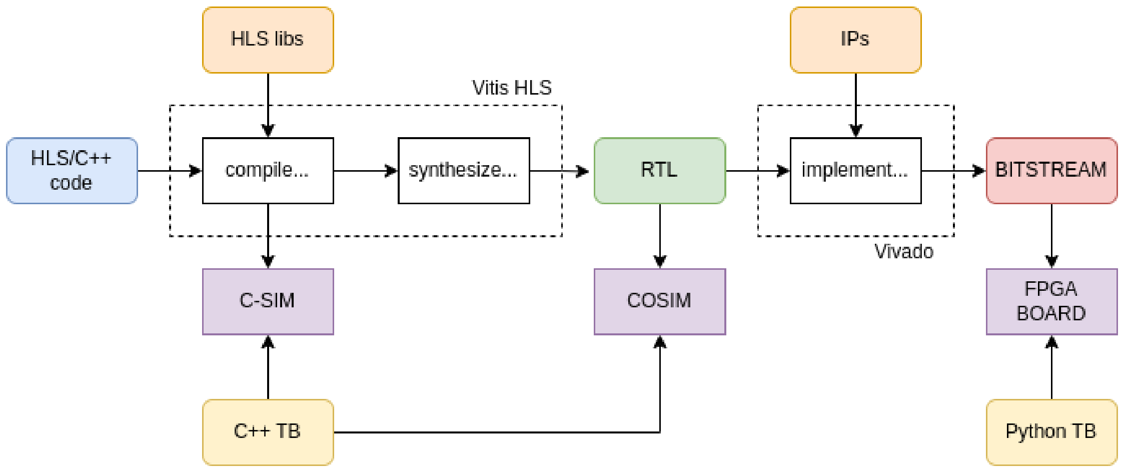
This paradigm follows an approaches to problems sugin a methodology similar to OS development one (decompose an algorithm in tasks and channels). Even this paradigm comes with its realated problems, for example deadlocks, the main advantage is represented by the resulting generalized view of an architecture which a designer can benefit from. To control the resulting architecture, at lower levels, **#pragma** directives are used which are written like other C language preprocessor directives, thanks to pragmas is possible to gain control of resulting parallelism.

Two main pragmas that can be applied to for loop are:

- **Unroll:** all iterations of the loop are executed in parallel by the RTL.
- **Pipeline:** all iterations still in sequence, but if one iteration requires more than one clock cycle, the resulting digital circuit is pipelined to manage one input data at each clock cycle.

Design workflow

Following figure shows typical workflow when designing an architecture using Xilinx tools, which also is the workflow used in this experiment:



The given illustration show how the workflow is split using two tools: Vitis HLS, that effectively enable the use HLS in designs thanks to it capability to convert C++ source code in RTL description, the resulting output is a collection of Verilog or VHDL files describing tasks, channel and top-level level block.

Vivado is not only used for this workflow but is the default AMD tool to implement projects described in HDL on a specific Xilinx board, the resulting output will be the bitstream to be loaded into FPGA.

Testing

The steps to be followed during HLS development are:

1. **Write C++/HLS** using any IDE actually used for software development.
2. **Compile** and **run C-SIM (C simulation)** which warrantee that, at least, the algorithm is logically correct and works, if not return to step 1.
3. **Synthesize** C++ always using Vitis HLS, then, once obtained the RTL, **run a co-simulation (COSIM)** to test the compliance between RTL and C++ code, if deadlocks or stalls occur during this kind of simulation , returning to step 1 is required, since these stalls certainly will appear during next steps.
4. **Run the implementation** that will map the RTL into available FPGA block, will configure all AXI interfaces and clock sources, this step require issuing a Vivado project to set these additional characteristics and check connections between FPGA, CPU and other devices.
5. **Upload bitstream** to FPGA and start hardware test using Python overlay, then **download results** from board to check compliance.

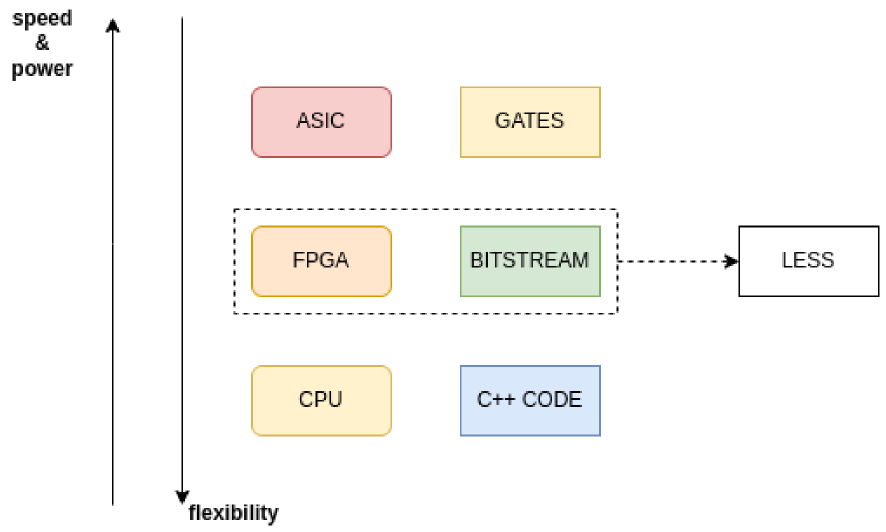
Chapter 2

The LESS graph isomorphism algorithm

Algorithms like MJW are already used in their CPU variants, the inconvenient is that these algorithms are very slow and consume a huge amount of power. We might think to use GPU which, thanks to their parallel architecture, can speed up the process, but the energy required to accomplish the operation might be higher than CPU one.

Another idea could be using an ASIC which integrates all logic circuits needed to carry out all steps of MWJ and, since they're specially realized for a specific task, high speeds can be obtained, but the inconvenient is that a single modification would require to realize the circuit again. FPGA is a good compromise between a custom hardware and flexibility, even they're slower than ASICs, and lower power consumptions can be achieved.

FPGA is rarely considered, excluding few exceptions like FAST [7]. All the already existing algorithms have as bottleneck the time spent to access data which aren't adjacent: neither locally or spatially [4], this is the main motivation behind the development of LESS architecture.



2.1 Introduction to LESS

The FPGA architecture considered in this thesis is LESS (**L**ow-power **E**nergy-efficient **S**ubgraph **Is**omorphism). The entire project is developed in HLS (High Level Synthesis), a paradigm developed by AMD for their Xilinx FPGA boards consisting in writing C++ code which is transformed in RTL, then the flow is the typical one of all HDLs.

LESS relies on tables mechanism to speed up the MWJ process, each edge has 4 values (starting node ID, ending node ID, starting node LABEL, ending node LABEL), when a candidate solution is being built the algorithm searches for edges which typically cover these requisites:

- It has a certain starting node ID and/or a specific ending node ID
- It has a certain starting node LABEL and ending node LABEL

If no tables are used, the process needs to read all the data edges resulting in a very time consuming operation, so these edges must be divided in different blocks into memory and the process must know where to search for an edge given candidate nodes/solution.

To achieve this improvement LESS is composed by two parts:

- **Preprocess:** read querygraph and datagraph to build tables
- **MWJ:** tables are accessed to build solutions

2.2 Preprocess

Preprocess can be divided in two steps:

- Tables building
- Tables filling

2.2.1 buildTableDescriptors

The first one consists in reading querygraph's edges and build tables accordingly, each edge has its starting node and its destination node, basing on some criteria like the nodes order given as input parameter, one node is chosen as indexing and the other one as indexed, a matrix of integers, **labelToTable**, given source and destination labels as coordinates, stores the table ID for each combination of indexing-indexed label which occurs at least one time during query edge scanning. **QueryVertex**, a vector of structures, stores, for each query node, tables for which these nodes are indexing or indexed.

Algorithmically, steps on querygraph are:

- Read each vertex and its related cardinality to fill **fromNumToPos** vector.
- Read all edges, for each one
  - Compare cardinality of start and end nodes, thanks to previously filled vector, the node which comes first will be the indexing, other one the indexed. Useful information to have a criteria to build tables.
  - Thanks to **numTables** counter and **labelToTable** matrix, mark that a certain table from this source node from this destination node exists, if not marked before. When assigning it, take into account which number of table will be when addressing it in memory.
  - Fill **qVertices** structure: increase the number of tables for which source node acts as indexing, then store table ID in the related vector of structure, does the same for destination node in terms of table for which this node acts as indexed.

2.2.2 fillTablesURAM

The second step consists in reading datagraph's edges to be stored in their belonging tables. An array of **AdjHT** structures stores where counters and edges are stored in DDR for each block. Initially, **countEdgesPerBlockWrap** and **storeEdgePerBlockWrap** split data edges in blocks, these two wrappers are built in similar ways: there are two tasks, fist one reads edges, accesses labelToTable matrix, and stream the tuple containing edge itself, the destination table address if src to dst, dst to src or both combinations of labels are found in query tables, second inner task receives tuples and, with the help of a little cache, stores edges in the correct block. Wrappers differs only because second inner task counts edge per block in the first case and store edges in a common DDR space which is indexed by offsets calculated between the calls of these two wrappers.

The **blockToHTB** function plays the role to transfer the information from blocks to AdjHT structures, then **writeBloom** write blooms which help MWJ to find out which set of edges has less collision (is important to stress on the fact that this is an heuristic assumption relying on some bit filled by hashes).

The overall algorithm of this second part:

- For each edge of datagraph:
  - Calculate hash on both start and end nodes.
  - Use labels of both nodes and labelToTable matrix to determine if tables for one or both directions exist.
  - Calculate addresses from hashes for both direction edges.
  - Use BRAM blocks to count edges.
- Convert all BRAM blocks to offset.
- A second cycle over edges:
  - Calculate hashes.
  - Determine if tables for one or both directions exist.
  - Calculate addresses from hashes for both direction edges.
  - Get offset from BRAM blocks
  - Store edge to a portion of DDR at this offset.
- Convert offsets from blocks one to tables ones and store starting addresses to hTables structures, store edges and counter in the htb_buff DDR space taking into account table number and block inside a table.
- Write blooms after edges have been correctly split in tables.

Once these macro steps are completed, another function propose starting nodes for MWJ and these nodes are stored in another portion of DDR.

The methodology to choose the starting candidates nodes is:

- For each table for which fist node is indexing:
  - Get offset and number of edges
  - If the this number is minimum select this table
- Propose one indexed node as start candidates for each hash set of edges.

2.3 MWJ

The entire chain begin from the fetching of partial solution from FIFO, but, in order to start the process, this FIFO must be filled with initial data, **mwj_assembly** has the role to read a starting candidate from ones proposed by preprocess, wait for verification of all partial solution and propose next starting candidate, a stop signal is provided to fifo after all starting candidates have completed their solutions expansion and verification.

Before getting into next steps it's fundamental to clarify how nodes are stored in the FIFO, **radix** are the nodes which has been verified as suitable for a partial solution, an **extension** is a node which has to be verified before it can be appended to the partial solution. To distinguish a radix from an extension, the MSB is used, an example of FIFO output is:

```
0x80000002, 0x80000001, 0x00000003, 0x00000005
```

This stream means that **nodes 2 and 1 has been verified**, **nodes 3 and 5 has been proposed as third nodes** for two partial solution, **{2,1,3}** and **{2,1,5}** will be the radixes after validation of third node.
FAKE_NODE and STOP_NODE, which are 0xFFFFFFFE and 0xFFFFFFFF, are special radixes which are used to stream single node partial solutions at begin (fake node + first extension) and to stop the whole pipeline (stop node).

Given a stream, another task, **mwj_edgebuild**, accesses to indexing structure built in preprocess and streams out tuples to represent all query edges which has the extension cardinality as indexed node, **mwj_findmin** selects the one with less collisions (to speed up next verification tasks) using the auxiliary bloom filter (filled in preprocess stage), then counters are accessed and all candidate edges are fetched from DRAM, mwj_homomorphism trashes homomorphism cases.

The steps for this first branch are:

- Read partial solution nodes until an extension is reached.
- Take the next positional node in the query order.
- For each table where this node is indexed:
  - Get table number and its related positional indexing node.
  - Use position to determine in this solution which node actually is acting as indexing (thanks to a local memory for previous nodes).
  - Read bloom at address given by table number and indexing vertex hash.
  - Calculate bloom fullness for each tuple.
- Select the tuple for which its bloom has minimum fullness.
- Access hTables and DDR portion to read counters.

- Obtain start and ending addresses where edges are located.
- For each edge in each row:
  - Only if bloom admit possible presence and indexing vertex is correct.
  - Verify if indexed node of this edge is already present in the current partial solution, in this case trash it (homomorphism).

The tasks from **mwj_tuplebuild** to **mwj_verify**, similarly to precedent ones, reads all edges sets, complete intersection and verification, then, the **mwj_assembly** checks if the solution is partial or full, in the first case it's written back into FIFO (only changed data is written), otherwise the counter of matches is increased and partial data are discarded.

Following illustration gives an overview of tasks and channels:

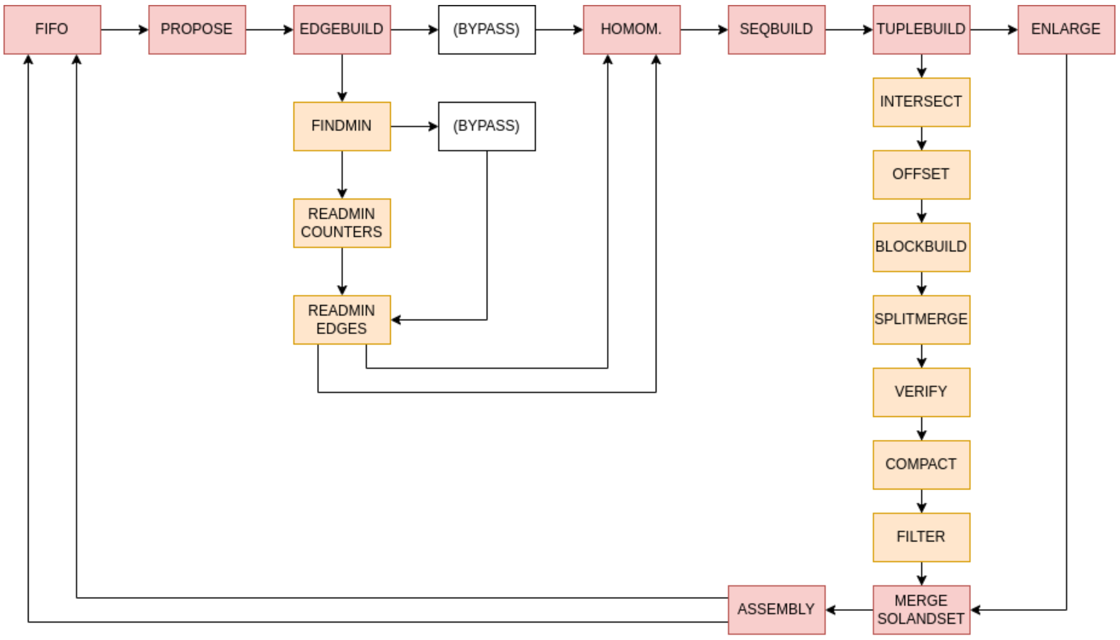


Figure 2.1. MWJ LESS core

Partial solution data are transferred between red highlighted blocks, assembly is linked to FIFO using two stream, one for actual data, other one for stop signal. Auxiliary tuples are streamed among orange blocks, the importance of this visual schematics is the evidence that immediately comes out: there are two chain to parallelize, first one from EDGEBUILD to HOMOMORPHISM, second one from TUPLEBUILD to MERGE SOLANDSET. In which task data can be actually divided is described in the next chapter about duplication of tasks.

Chapter 3

Increasing parallelism

Modifying parallelism (x2 in this experiment) require two steps, first, since data have to be split in different memories, to change how edges are stored, second, doubling some tasks, at least the ones involving dataflows having DRAM accesses.

Preprocess is responsible of edges storage in DRAM, since it builds tables and, for each data edge, it selects which will be its storage zone.

3.1 Preprocess

The part which consists in table building hasn't been modified since it models the querygraph itself and it doesn't carry any information on data edges.

All structures concerning the second macro step has been doubled (AdjHT, blocks vectors, the common edges space and the blooms), the hash MSB is used to establish which of two side is the destination of an edge. To support these changes, the tuples streamed between first and second task of count and store wrappers is extended with one extra field (in this case is only 1 bit since the parallelism is x2), the second task receives both pointers of blocks and edge memories, thanks to this extra field it can decide where to store the edge and which blocks vector has to be updated. All the next tasks as blockToHTB, writeBloom and start candidates proposals are simply doubled since all edges and their related information are fully split by the wrappers.

Concerning **mwj_batch** (start nodes proposal task), candidates memory is kept single since it stores only nodes (hash independent) and not edges, second call is done by simply pushing other nodes after ones which where pushed by first call.

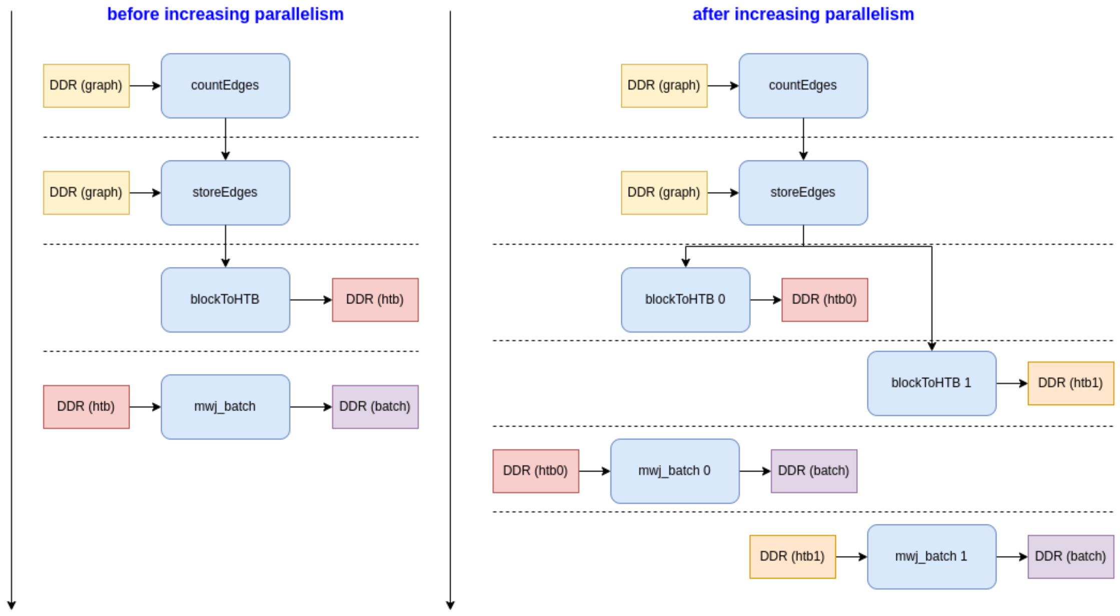


Figure 3.1. Overall preprocess transformation

3.2 Edgebuild branch

This branch plays the role to propose all the edges sets involving the next node to be mapped for a solution, propagate the only one (minset) chosen by findmin task which requires bloom filters, path is split by **mwj_findchannel** until homomorphism task is reached, then **mwj_merge_h** joins both task's outputs.

3.2.1 Enlarge solutions

Partial solutions are stored in the FIFO in a compressed format: this means that two solution having both {A,B,C} as radixes and different extensions (D and E) are streamed out as {A,B,C,D,E} instead of {A,B,C,D,A,B,C,E} by **mwj_propose** which only translate integer format of FIFO in vertex_t node structure.

The solution stream is kept compressed until **mwj_tuplebuild** task in the original architecture, then a task, called **mwj_enlarge_sol**, decompress it and mark complete solutions to be counted by assembly task. After doubling streams in the edgebuild branch, the challenge was to keep the alignment between extension nodes and their related findmin tuples due to channel splitting, to achieve the desired alignment decompression taks (mwj_enlarge_sol) has been divided in two parts: the decompression itself and the complete solution marking. First one has the original name and has been moved between propose and edgebuild tasks, the second one it's called **mwj_fulldetect** and maintain its original position (after tuplebuild).

Once solutions are decompressed, entire packet and it's related findmin tuple are aligned since they're both streamed in their belonging channel.

3.2.2 Edgebuild task and solution bypass

There aren't relevant changes to this tasks since its role is to read QueryVertex structures, hashing the vertex to be mapped and calculate the table address of each proposed sets of edges and the logic behind does not depend on channel to be chosen, due to this reason the process hasn't been split. Minor changes includes remotion of some fields and initializations from tuples which now aren't needed (thanks to decompression applied by precedent task and due to the split of partial solution streams).

Task which does the bypass of solutions has been removed since before doubling path the output stream of edgebuild was directly going to homomorphism task trough this bypass (which was need to avoid deadlock), now the solution is forwarded trough findmin taks also to keep the alignment between solutions and tuples in the channels.

3.2.3 Findmin task

Since blooms has been doubled in preprocess stage and findmin select the minset thanks to these filters, the needed modification is hashing indexing vertex of input tuple, take out its MSB and access to the correct of two bloom spaces.

3.2.4 Findchannel

mwj_findchannel has 3 input streams (solution, findmin tuple, bloom filter), every output stream from precedent tasks is forwarded there to keep the data alignment and, every time a tuple is received at input from a blocking read, it's treated differently:

- **if is stop:** in this case stop node and stop tuples are propagated in both channels to send an "end-of-stream" signal to both channel's tasks.
- **if isn't stop:** MSB of indexing vertex's hash is used to decide the channel, tuple is sent here, then its related filter and solution's nodes packet (vertex_t stream) is sent on the same channel, so data can be processed by next tasks (until homomorphism process) in the same way they did before duplication since each instance is called giving to it the pointer of correct edges zone.

3.2.5 merge_h task

The output of each homomorphism instance is an hybrid stream, solution nodes and minset are sent in a joined sequence, **mwj_sequencebuild_t**. Both task's streams are sent to an additional task, **mwj_merge_h**, which tries to read both input channels (non blocking), when one of these two ones has available data, the function continue getting data from this channel until the expected stream pattern is fully covered (partial solution nodes, minset, vertices to verify), so process will wait for first last flag, read an additional node (except when stop node occurs) which is minset, and will wait again for last flag. Both non blocking reads attempt occurs until their related channels stop node is received, after both stops occurred task streams its stop node and end its execution.

3.2.6 Overall transformation

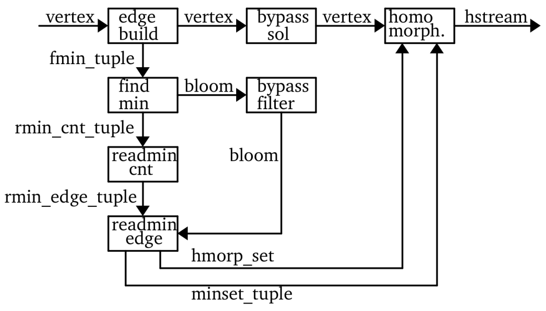


Figure 3.2. edge build branch before duplication

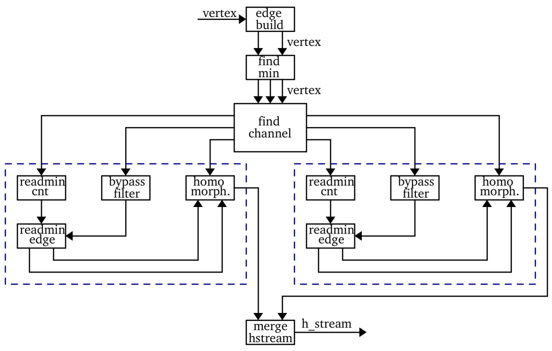


Figure 3.3. edge build branch after duplication

3.3 Tuplebuild branch

This second branch reads remaining edges sets to verify the incoming vertices from precedent tasks, this time changes involved also stream format modification. Without additional tuples, splitting paths resulted in a wrong behavior of verification task (if last tuple to verify arrived before other tuples carried in the other channel solution risked to be marked as false valid, on the other hand, the remaining tuples were mixed with ones of the next solution making it as false negative).

3.3.1 Tuplebuild changes

Solutions output has kept as it was, the intersect_tuple stream has been doubled, each time tuplebuild reads a tuple there 4 cases:

- **stop node:** stop tuples are propagated on both tuple output channels which are received by their chains and task ends.
- **solution:** it is propagated in the single solution output channel.
- **last_set:** it is propagated on both tuple output channels
- **else:** it is a tuple which has to be sent on the belonging channel, so the vertex to verify is hashed and the MSB is used to decide destination channel.

Concerning the 4th case, **extra_tuple** flag is asserted when last_edge flag of input tuple is set, this makes the function to stream an additional last_set tuple on both output channels before reading next input, to distinguish this last_set from the real last_set, the pos member of structure is used (0 if is the real last set, 1 if is this padding workaround).

3.3.2 Intersect, verify and compact

Tasks from intersect to compact integrates minor changes to treat the last_set flag correctly (it is considered as before if pos is 0 else it's simply ignored and propagated).

3.3.3 Merge assembly tuples

After intersection and verification, the tuple streams are joined by another task, called **mwj_mergeasmset**, non blocking read attempts on both input channels occurs, when a tuple is available, there are 4 different cases:

- **Regular tuple:** just send it to output.
- **Last edge:** copy it to a local memory structure.
- **Last set:** evaluate if

  - **pos=1:** it's the padding tuple, so wait for it on the other channel, if it's second time stream out the last edge which was stored before.
  - **pos=0:** it's real last set, just send it to output

  then: unlock reads from both input channels

Output tuples are sent to filter, merge sol and set chain, which hasn't been modified.

3.3.4 Overall transformation

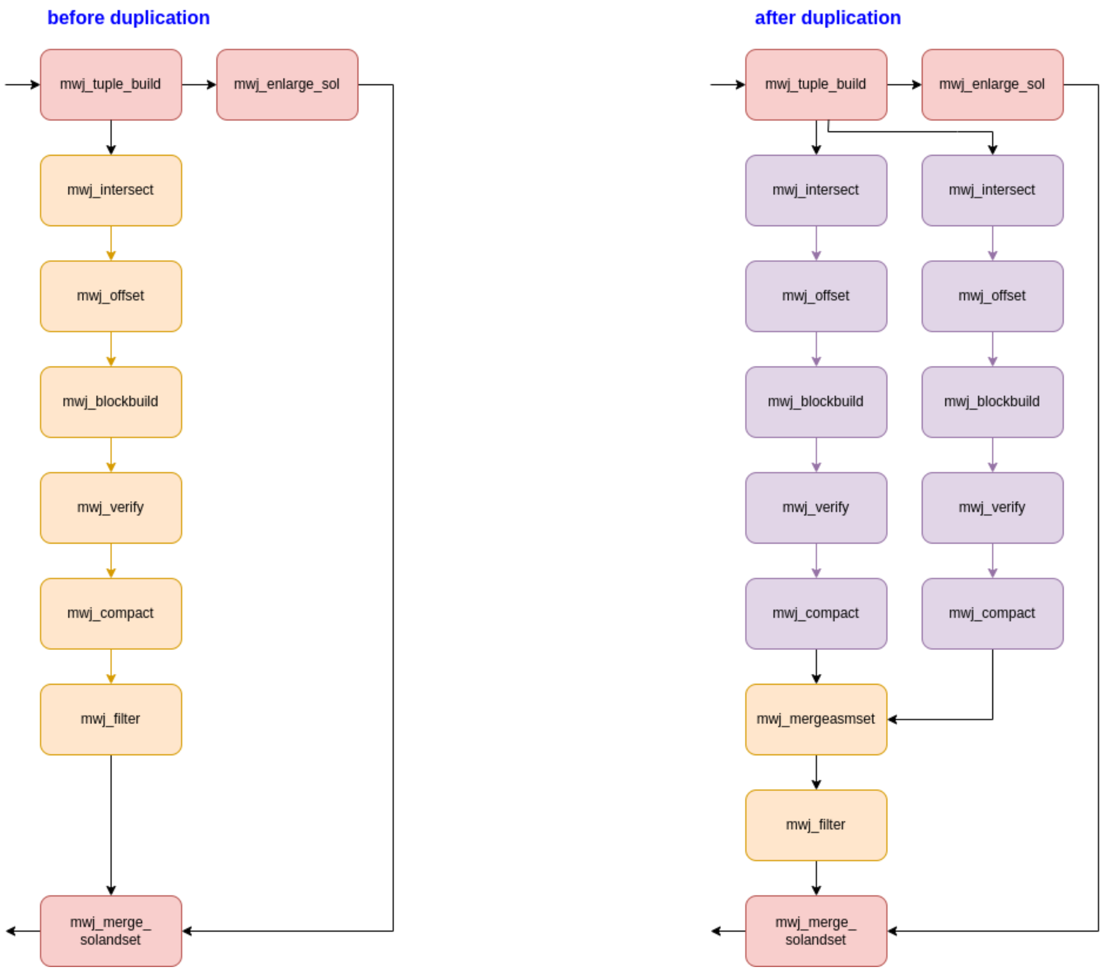


Figure 3.4. tuplebuild branch changes

Chapter 4

Results and considerations

Real-world database (for example from GitHub and Wikipedia) have been used as testing databases, they're collected by the reference: [3]. Golden results (number of expected matches passed to testbench) have been generated with the help of a golden model base on NetworkX [1]

Before analyzing obtained results, **one consideration must be done on the hardware that has been used during different tests : an FPGA board** (Xilinx Kria K26) having only the DDR and not the HBM banks.
HBMs (High Bandwidth Memories) are the required hardware to be used during the tests in order to notice performance improvements, if they are not used, the duplicated architecture can only prove the performances loss caused by modifications and extra blocks added to LESS architecture.

At each stage, 127 queries are executed, the number of matches varies from some tens of thousands to 1 billion thus permitting to test the core at different situations. These 127 test have been executed 4 times during development:

1. Original LESS architecture
2. After first branch duplication
3. After first and second one
4. After preprocess has been modified

During comparisons these 4 iterations of test well be cited as {v1,v2,v3,v4}.

To compare performances only actual query time will be considered, the main reason behind that is the possibility to omit this time as the query time increases, for example, some queries took over 1000 seconds to complete query while preprocess only took less than 500 milliseconds.

4.1 First stage duplication

At this stage the **v2** tests have been performed, changes involve the branch including tasks from **EDGEBUILD to HOMOMORPHISM**.

This step is the main source of performance losses:

- The **earlier enlargement of compressed solution** nodes in the streams causes more data to be streamed, thus increasing channels fullness and blockage of tasks while performing blocking readings or writings.
- Removing bypass tasks and letting partial solutions to be carried by findmin and following tasks caused the creation of additional local memories elements and loops, which not only add overhead in terms of processing times, but also **caused the pipeline pragma to be not fully applicable** to entire task without the risk of deadlocks. The most restricting requirement is the alignment of solutions and tuples that requires one loop before another one, thus making the concept of pipelining not easily applicable.
- Last, but not the less important, **DDR time bottleneck is increased by the introduction of other arbiters** (the Smart AXI controllers) to manage the DDR access, in simpler manner: reading two data (basically the edges) from single channel is faster than reading these two data from two different channels linked to an arbiter connected to same memory as before.

The ratio between execution time of this new version and the original architecture one gave these statistics: 0.82 as minimum, 2.09 as maximum, 1.37 on average.

Next figure better demonstrates the resulting trend of this first comparison:

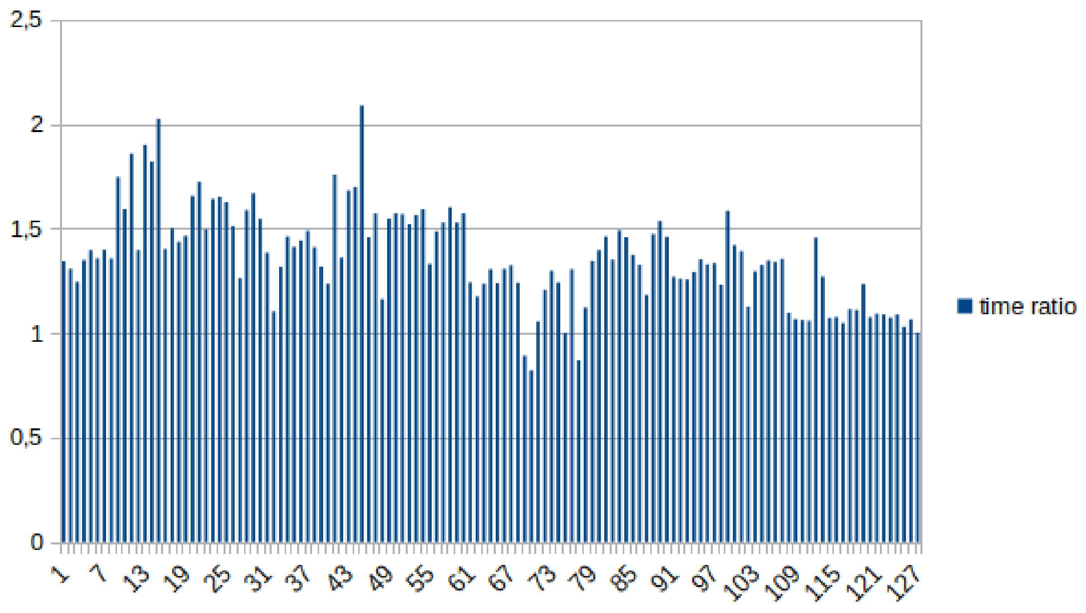


Figure 4.1. query_time(v2)/query_time(v1), preprocess excluded

There are few tests which require the double amount of time, even if the architecture is implemented using boards having HBMs, these test most probably cannot benefits from performance improvements, if parallelism is of second order.

Considerations aren't different for other tests which obtained time ratios from 1.5 to 2, doubling memory asymptotically can improve performance of a factor 2X, but this coefficient may be real only if these conditions occurs:

- Edges are exactly divided in two parts between two memories. (actually these two parts are **almost** equal).
- Each edge reading from memory 1 must happen when it also happens for memory 2. (Actually there may be intervals when only 1 memory is accessed).
- Memories must be ideal and have the same delay.

These few analyzed cases may benefit from parallelization if the degree is above 2, if time ratio isn't also increasing linearly with parallelism.

4.2 Second stage duplication

This second part involves branch from **TUPLEBUILD** task to **MERGE SOLANDSET**, losses encountered, applying this modification, are caused by:

- As before, other arbiters slow down access to a single DDR.
- Introduction of extra tuples to warrantee data alignment before filtering slow down entire flow of data trough streaming channels.

First comparison is made between **v3** (this version having both branches duplicated) and **v2** (which has only first branch parallelized), statistics in terms of (min, avg, max) are 0.77, 1.24, 2.23, in details:

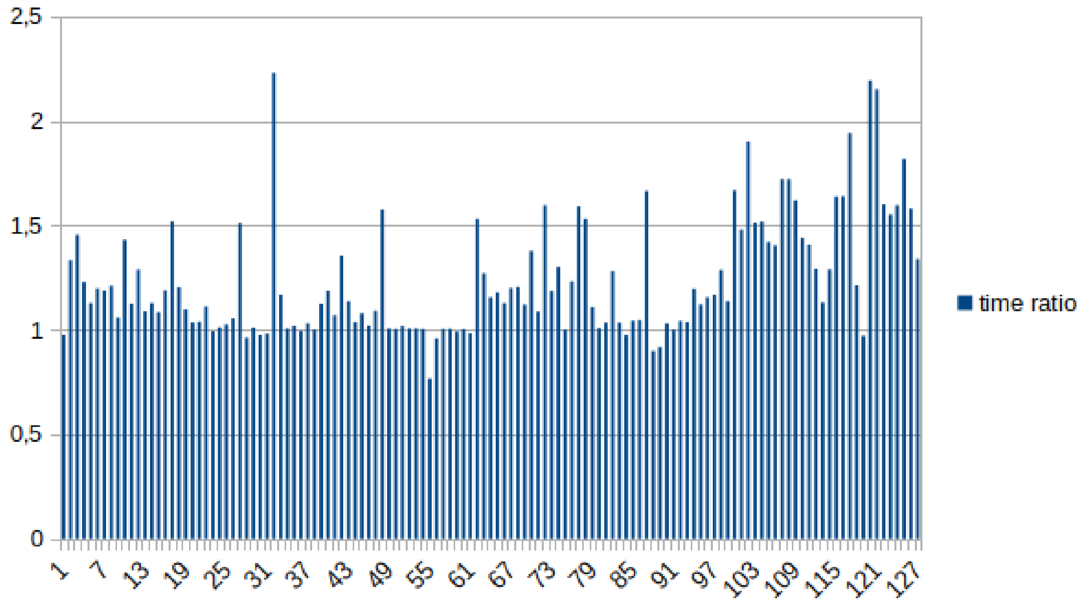


Figure 4.2. query_time(v3)/query_time(v2), preprocess excluded

The average ratio is lower than precedent one, what is different is the previous pipelining issue reported for FINDMIN, what is in common consists of the AXI arbiters which seems to slow down each branch requiring 20% of extra time.

Next chart reports comparison made from original architecture to this version:

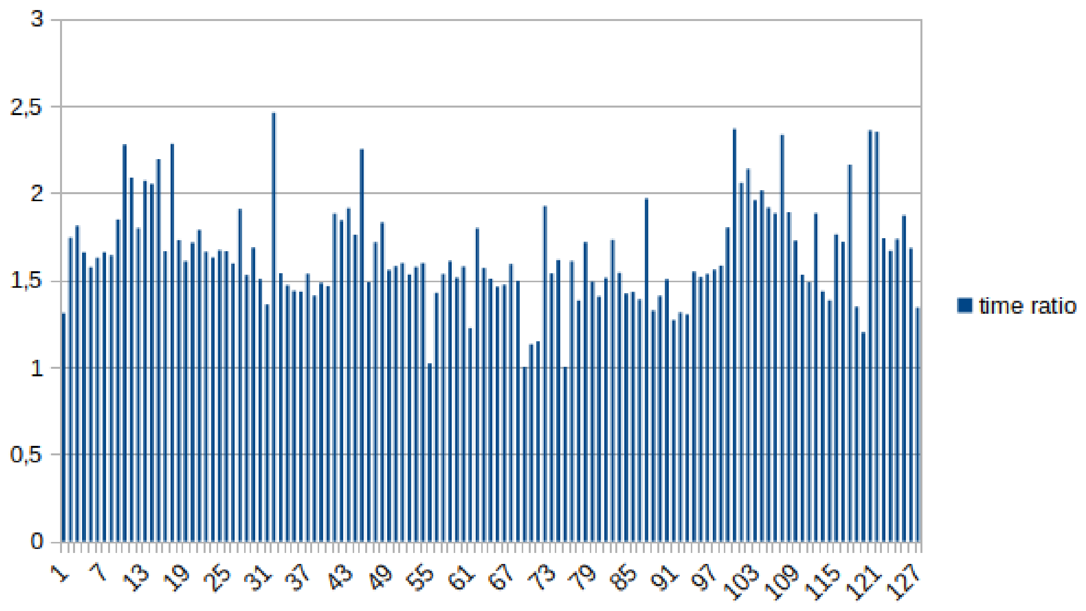


Figure 4.3. query_time(v3)/query_time(v1), preprocess excluded

Statistics, in this scenario, are (0.9995, 1.6541, 2.4611), which are reported here using two extra digits, to evidence that minimum case almost approached to 1.

It's also crucial to notice how in both transitions (v1 to v2 and v2 to v3) there are cases where time increases by 2 or more, while there isn't any case from v1 to v3 in which time increase by 4 and, at same time, v2 to v3 comparison obtained a minimum time shrink lower than v1 to v2 one.

Both facts confirms that there is a large budget in terms of time (that can both increase or reduce) due to DDR access conflicts: actually on second stage a lower minimum ratio is obtained since both v2 and v3 has these access bottlenecks and any of other characteristics that might influence time alignment between memory requests can lead to a large difference during servicing them.

4.3 Memories duplication

Any of changes that involves tasks in this part can only influence preprocess time if the architecture is strictly analyzed in terms of algorithm, any improvement or loss introduced during the MWJ core execution is only due to the same reasons that affected all previous phases: the Smart AXI arbiters.

Another factor which may be decisive is the sizing of h1 and h2 (the width of hashes for indexing and indexed vertices), the Python testbenches, for all these 4 groups of test, has a basic algorithm to determine which (h1,h2) are better for the specific query that has to be processed (this algorithm has been changed a bit when preprocess has been modified to avoid overflows during block splitting).

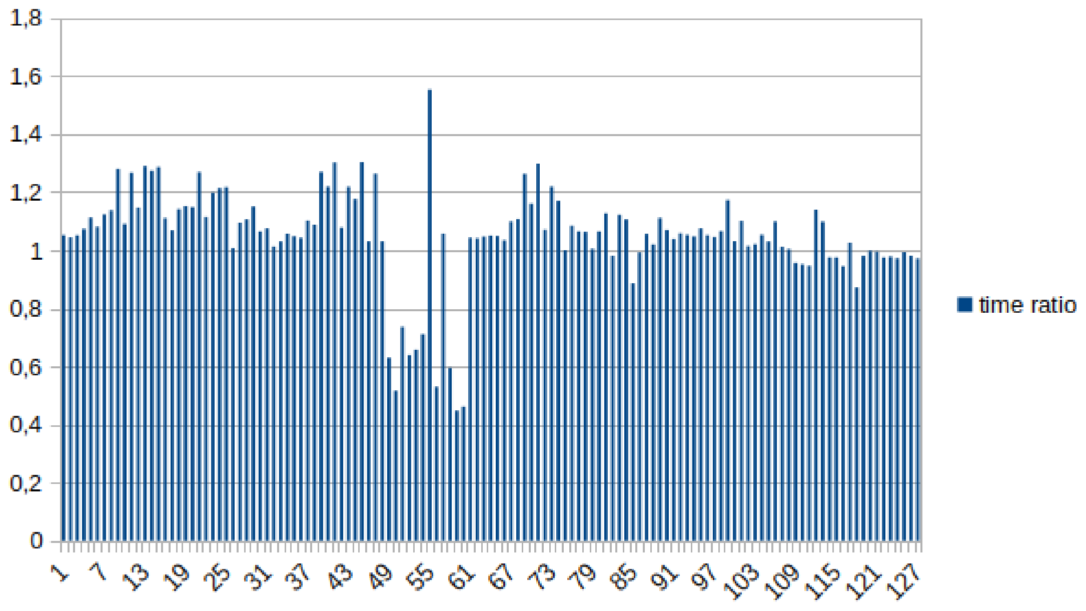


Figure 4.4. query_time(v4)/query_time(v3), preprocess excluded

The statistics for this comparison are (0.45,1.05,1.55), everything is confirming previously made deductions, the counterintuitive results comes in following chart:

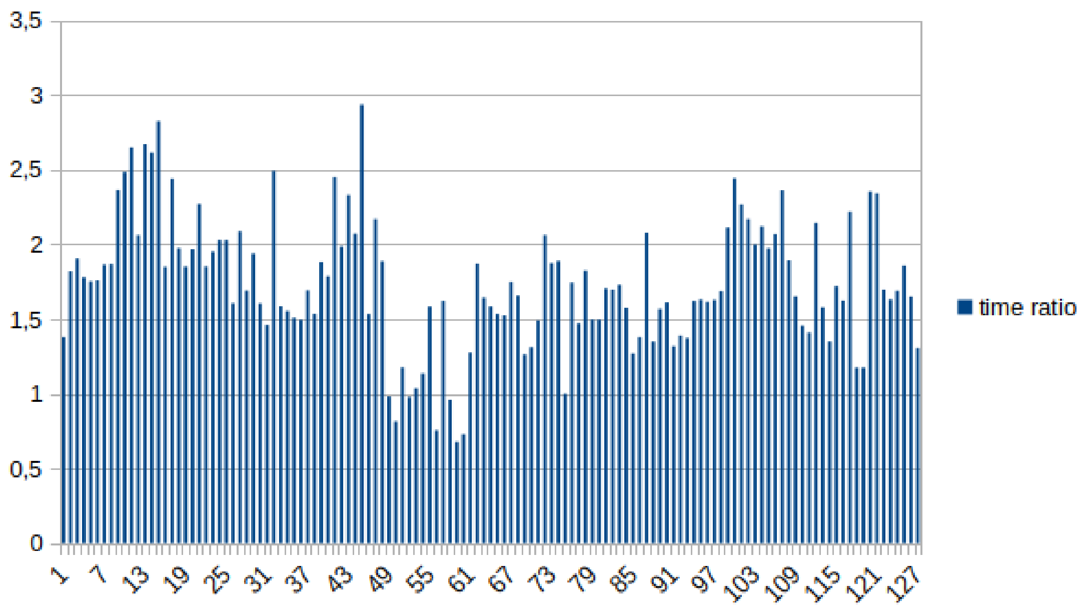


Figure 4.5. query_time(v4)/query_time(v1), preprocess excluded

Statistics for this last comparison (the final one) are (0.68,1.75,2.94). What is totally misleading, more than before, is the minimum ratio. It's virtually unrealistic that there are cases from original architecture to last one giving performance improvements, from what a query can benefits if there are only losses introduced (if the architecture hasn't moved to HBMs)?

Answer came out, after analyzing v1 and v4 tables of results, all of them share the characteristic to is that h2 has been reduced in the last test (for the reason mentioned before regarding the algorithm to calculated optimal one), and most of them has a reduction of 2 bits.

If the same analysis is conducted from table v3 to v4, there is time reduction also in cases that doesn't have any changes in h1 or h2, in this scenario time reduction is caused, not only by the possible shiftings in terms of requests alignment but also by redistribution of AXI ports made in last version.

Following figures shows connections of both orginal and final architectures:

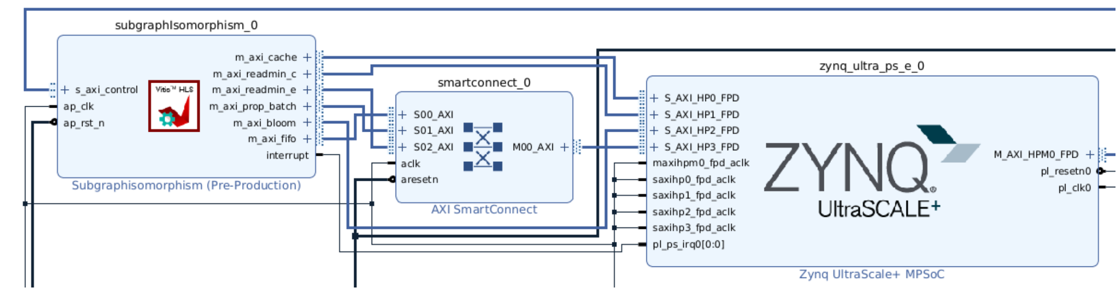


Figure 4.6. AXI connections of original architecture

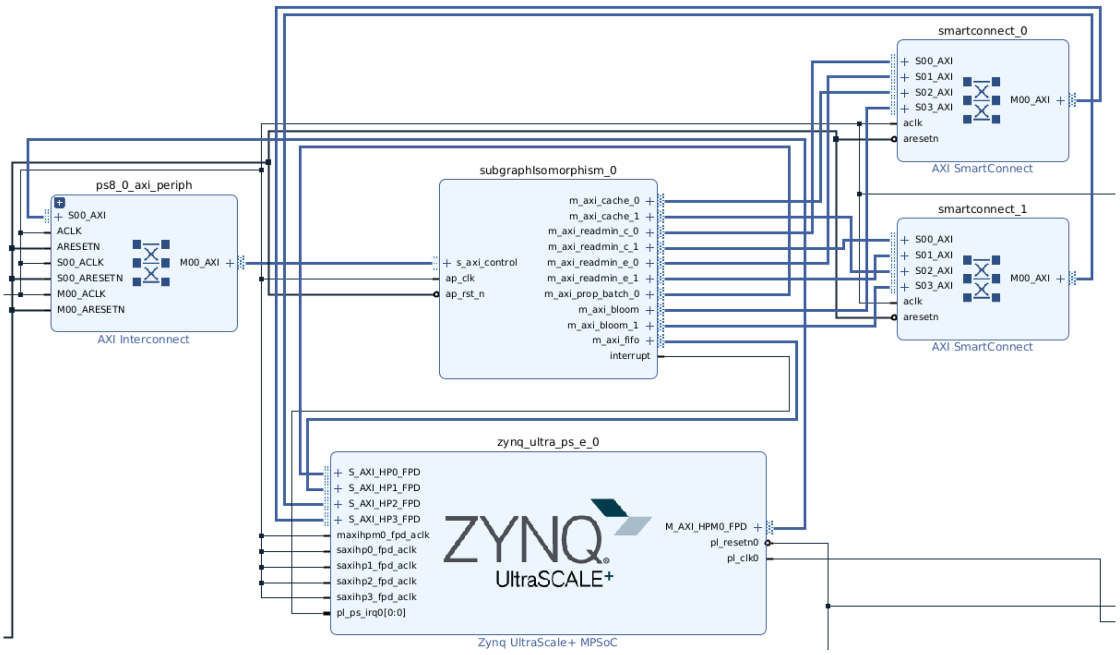


Figure 4.7. AXI connections of final architecture

4.4 Further improvements e considerations

The new architecture must be tested on an FPGA board having enough HBMs to support parallel read requests to evaluate the actual gain of this experiment, however, other suggestions and improvements suggested in ext paragraphs might speed up the architecture even more.

4.4.1 Consider 4th degree parallelism

By looking the average of final comparison time ratio (1.75) and assuming that the further changes to support the transition from 2nd degree to 4th one won't introduce a large amount of performance loss as happened in the case of 1st to 2nd degree, the increment of parallelism can help the architecture to benefit more than parallelization thus enhancing a gain in terms of speed which passes the performance loss by changes that have been performed on streams.

4.4.2 Reorganize pipelines and stream format

As stated before, first branch isn't fully pipelinable after solutions stream has been redirected inside same task of tuples: to overcome this issue a possible solution is to use same stream for tuples and vertices using extra flags to distinguish them.

Appendices

Appendix A

Input data

A.1 Used datagraphs

```
dblp.RM.csv : 317080 vertices, 1049866 edges
enron.RM.csv : 36692 vertices, 183831 edges
github.RM.csv : 37700 vertices, 289003 edges
gowalla.RM.csv : 196591 vertices, 950327 edges
wikitalk.RM.csv : 2394385 vertices, 4659565 edges
```

A.2 Used queries

```
query0.RM.csv : 3 vertices, 3 edges
query1.RM.csv : 4 vertices, 4 edges
query2.RM.csv : 4 vertices, 4 edges
query3.RM.csv : 5 vertices, 8 edges
query4.RM.csv : 5 vertices, 8 edges
query5.RM.csv : 5 vertices, 8 edges
query6.RM.csv : 5 vertices, 7 edges
query7.RM.csv : 5 vertices, 8 edges
query8.RM.csv : 5 vertices, 6 edges
query9.RM.csv : 6 vertices, 8 edges
query10.RM.csv : 6 vertices, 8 edges
query11.RM.csv : 6 vertices, 9 edges
query12.RM.csv : 6 vertices, 9 edges
query13.RM.csv : 6 vertices, 9 edges
query14.RM.csv : 6 vertices, 8 edges
query15.RM.csv : 6 vertices, 11 edges
```

```
query16.RM.csv : 6 vertices, 8 edges
query17.RM.csv : 6 vertices, 10 edges
query18.RM.csv : 7 vertices, 16 edges
query19.RM.csv : 7 vertices, 17 edges
query20.RM.csv : 7 vertices, 15 edges
query21.RM.csv : 7 vertices, 15 edges
query22.RM.csv : 7 vertices, 16 edges
query23.RM.csv : 7 vertices, 16 edges
query24.RM.csv : 7 vertices, 15 edges
query25.RM.csv : 7 vertices, 17 edges
query26.RM.csv : 7 vertices, 14 edges
query27.RM.csv : 8 vertices, 26 edges
query28.RM.csv : 8 vertices, 23 edges
query29.RM.csv : 8 vertices, 22 edges
```

Bibliography

[1] NetworkX documentation, 2022. URL https://networkx.org/documentation/stable/reference/algorithms/isomorphism.vf2.html.

[2] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. *W. H. Freeman and Co.*, page (cit. on p. 2), 1979.

[3] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection. URL http://snap.stanford.edu/data.

[4] Johannes De Fine Licht Tal Ben-Nun Maciej Besta, Dimitri Stanojevic and Torsten Hoefler. Graph processing on fpgas: Taxonomy, survey, challenges. page (cit. on p. 17), 2019. doi: 10.48550/ARXIV.1903.06697. URL https://arxiv.org/abs/1903.06697.

[5] Ajeet Grewal Siva Gurumurthy Volodymyr Zhabiuk-Quannan Li Pankaj Gupta, Venu Satuluri and Jimmy Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proc. (VLDB)*, 2014.

[6] Xilinx. Vitis high-level synthesis user guide. URL https://docs.amd.com/r/2021.2-English/ug1399-vitis-hls.

[7] Xuemin Lin Shiyu Yang Lu Qin Xin Jin, Zhengyi Yang and You Peng. Fast: Fpga-based subgraph matching on massive graphs. page (cit. on p. 17), 2021. doi: 10.48550/ARXIV.2102.10768. URL https://arxiv.org/abs/2102.10768.