Accelerating Regular Path Queries using FPGA

Kento Miura
Dept. of Computer Science
University of Tsukuba
1–1–1 Tennodai, Tsukuba
Ibaraki 305–8577 Japan
miura.k@kde.cs.tsukuba.ac.jp

Toshiyuki Amagasa
Center for Computational Sciences
University of Tsukuba
1–1–1 Tennodai, Tsukuba
Ibaraki 305–8577 Japan
amagasa@cs.tsukuba.ac.jp

Hiroyuki Kitagawa
Center for Computational Sciences
University of Tsukuba
1–1–1 Tennodai, Tsukuba
Ibaraki 305–8577 Japan
kitagawa@cs.tsukuba.ac.jp

ABSTRACT

This paper proposes a scheme for accelerating regular-path queries (RPQs) for directed edge-labeled graphs using an FPGA. Graphs are quite useful to represent various types of relationships among different entities and have been used in diverse fields, such as social networking analysis, linked open data (LOD), and bioscience. RPQs are queries to retrieve pairs of vertices that are reachable through a path whose labels conform to a user-specified regular expression. Despite its importance and usefulness, RPQs have not been paid much attention. In this paper, we attempt to accelerate such queries using an FPGA (field programmable gate array). Specifically, we propose a pipelined process of RPQs by dividing a query into multiple stages thereby taking advantage of pipeline parallelism. Experimental evaluations show that the proposed accelerator achieves up to 23.6x faster for the small dataset and up to 4.61x faster for large dataset than the comparative method running on CPU.

1. INTRODUCTION

A graph is useful to represent different kinds of data in real-world applications in particular when one needs to represent relationships among different entities. Social networks, linked open data, and chemical compound networks are well-known examples.

Given a directed edge-labeled graph, regular path queries (RPQs) [10] are used to retrieve pairs of vertices that are reachable through paths whose labels conform to the user-specified regular expression. RPQs are useful in such applications where pairs of vertices that are connected via a specific path. For example, a *meta-path* is represented as a regular path, and its occurrences are extracted from a graph for subsequent tasks [22, 23].

As can easily be imagined, processing an RPQ is expensive in particular when processing large graphs, while the demand for high-speed processing of large graphs is increasingly growing. FPGA (field programmable gate array) is a possible way to accelerate RPQ processing. FPGA is a device that allows users to implement arbitrary logic circuits by programming dynamically and has recently been applied to various problems, such as query processing relational database systems [8], deep neural networks [26], and graph processing [11]. Despite its low frequency compared to CPUs, if the device is optimally configured, it is possible to outperform them.

In this paper we propose a novel scheme for processing simple RPQs using FPGA. To maximize the performance, given an RPQ, we divide it into multiple stages according to path steps, and process the query in a pipelined manner whereby we can exploit pipeline parallelism. We additionally propose an optimization of pipeline execution to improve the performance further. Experimental evaluations show that the proposed accelerator achieves up to 23.6x faster for the small dataset and up to 4.61x faster for large dataset than the comparative method running on CPU.

This paper is organized as follows. We describe preliminaries of this research in Section 2. Section 3 introduces related research works on FPGA-based accelerators and RPQ processing. We explain the proposed scheme in Section 4, and show the performance study in Section 5. Section 6 concludes this paper.

2. PRELIMINARIES

2.1 Regular Path Queries (RPQs)

Given a directed edge-labeled graph, a *regular path query* (RPQ) is a query to retrieve pairs of vertices that are reachable via such paths whose label sequences conform to the user-specified regular expression. More precisely, let $G = (V, E, \mathcal{L}, \lambda)$ be a directed edge-labeled graph where V is a set of vertices, E is a set of edges, and $\mathcal{L}$ is a set of edge labels, and $\lambda : E \to \mathcal{L}$ is a mapping from an edge to an edge label. Then, a *regular path query* RPQ R is defined as follows:

$$R ::= \epsilon \,|\, \ell \,|\, \ell^- \,|\, R \circ R \,|\, R \cup R \,|\, R^{i,j}$$

where ϵ is an empty path; ℓ and $\ell^- \in \mathcal{L}$ are a forward and backward navigation, respectively; $\circ$ and $\cup$ are path composition and disjunction, respectively; and $R^{i,j}$ is bounded path recursion. Given a graph G and a regular path query R, an evaluation of R over G, denoted as $R(G)$, returns all pairs of vertices $v, w \in V$ such that 1) there exists a path from v to w in G, and 2) the path expression from v to w

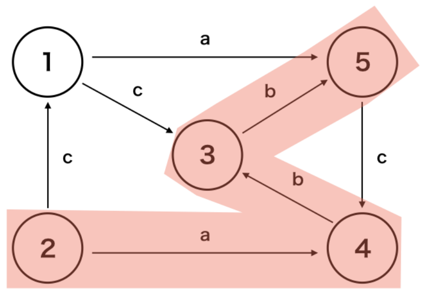


Figure 1: An example of RPQ ($R = a \circ b \circ b$).

matches to the regular path query R. For example, when applying an RPQ $R = a \circ a \circ b$ against a graph G_{ex} depicted in Figure 1, we get:

$$R(G_{ex}) = \{(2, 5)\}$$

In this example, the highlighted part is the result of the query.

In this paper we assume that RPQs are *simple*, i.e., RPQs only contain path composition operation for simplicity. Supporting more complected operations is our future work.

2.2 FPGAs

Field programmable gate array (FPGA) is a hardware where arbitrary logical circuits can be implemented dynamically by programming. Recently, the performance of FPGAs has been improving significantly, and one can implement more complex circuits. For this reason, FPGAs have been used in various fields, such as numerical calculation [20], image processing [18], etc.

In general, to program an FPGA, a programmer needs to use a dedicated hardware description language, such as Verilog [6] or VHDL [1]. In this case, the programmer is required to be familiar not only with the language but also with hardware device, which is considered to be a problem because the number of such experts are limited. Alternatively, we may use high-level synthesis (HLS) [9] whereby one can generate program circuits on FPGAs by writing a program in high-level languages such as C/C++ and OpenCL [3]. In this research, we implemented our circuit on the FPGA using this HLS.

3. RELATED WORK

3.1 FPGA-based accelerators

Recently, FPGAs have been used to accelerate different workloads in various problem domains. For example, Sidler et al. used an FPGA to speed up queries containing regular expressions against text strings in a relational database [21]. Ham et al. developed an FPGA accelerator, Graphicionado [14], to accelerate graph analysis, such as PageRank [19] and collaborative filtering [7].

In commercial systems, Netezza (now PureData System for Analytics [2]) from IBM achieved significant improvement on throughput using an FPGA.

3.2 RPQ Processing over RDF and Graphs

In the context of *Resource Description Framework* (RDF) and graph processing, RPQs are considered to be an important class of queries and therefore have been studied by many researchers. For example, PGQL [25] developed by Oracle and openCypher [4] support RPQs. Besides, Neo4j cypher [13] and SPARQL 1.1 [5] also support RPQs.

3.3 Efficient Processing of RPQs

There have been many research works that attempt to speed up RPQ processing. Their approach can roughly be categorized into automata-based [16] and index-based [12].

In the former approach, given an RPQ, the automaton that corresponds to the query is generated, and is applied to the graph being processed. In [16], more efficient search using automata is realized by dividing the graph by labels with low frequency of occurrence in the graph and reducing the search space. However, this method requires the existence of labels with low frequency of occurrence, and the performance depends on whether or not the labels can be selected appropriately.

To cope with the performance problem of automata-based approach, index-based approach have been studied. In *Path Index* [12], a dedicated index structure is proposed where all occurrences of paths up to length k is extracted and stored in corresponding indexes. Taking Figure 1 as an example, all paths up to length $k = 2$ is stored in indexes represented as tables in Table 1. When processing queries, if the given query is shorter or equal to k, the results can be obtained just by looking up the indexes; otherwise, the results can be obtained by joining multiple indexes; e.g., length 4 can be obtained by self-join of length 2 index. A drawback of this approach is that the size of index rapidly increases when increasing parameter k, while join cost dominates when processing long RPQs with indexes of small k value.

Table 1: Path Index

(a) k = 1

Path	Source	Destination
a	1	5
a	2	4
b	3	5
b	4	3
c	1	3
c	2	1
c	5	4

(b) k = 2

Path	Source	Destination
$a \circ b$	2	3
$a \circ c$	1	4
$b \circ b$	4	5
$b \circ c$	3	4
$c \circ a$	2	5
$c \circ b$	1	5
$c \circ b$	5	3
$c \circ c$	2	3

4. FPGA-BASED ACCELERATION OF RPQS

In this research, we propose a method to parallel processing of RPQs using FPGA. The process is divided into CPU and FPGA part; graph data is stored in disk and are sent from CPU to FPGA, and a pipeline-based parallelism is applied on FPGA for speeding up query processing.

4.1 Process Overview

Figure 2 shows an overview of the proposed method. Let us assume an RPQ $R = a \circ b \circ b$. The query processing is performed on both host (CPU) and device (FPGA) sides. As a preprocessing on host side, the graph is stored on the secondary storage (HDD/SSD) in such a way that the edges of the same label are stored in the same file. When processing a query, the host first analyze the query and determines the labels appearing in the query. Then, the host sends the query to the device for making it ready to process the query. Afterwards, the host sends the pieces of stored graph data to the device for actual query processing.

On the device side, our strategy is to divide the query into multiple stages according to the path steps included in the query, and perform multi-way join. More precisely, for query $R = a \circ b \circ b$, the edge table corresponding to the label a and b are fed into FPGA, and they are merged according to their connectivity. Finally, the result is sent back to the host side.

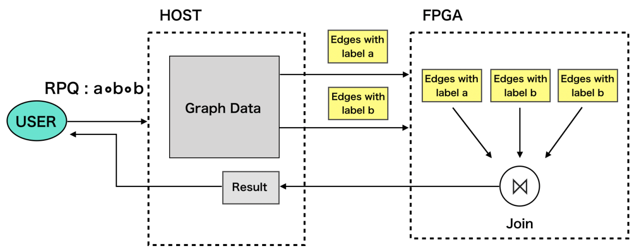


Figure 2: Storage format of graph data.

4.2 Graph Data Storage

As a preprocessing, the graph is divided into different pieces, and is stored in the secondary storage. More precisely, since we take an edge-wise join strategy to process an RPQ, we divide the graph according to edges with the same label (e.g., a, b, etc.) Also, to make the edge-wise join easier, we store the same set of edges (with a label) in two ways; i.e., sorted according to the ascending order of source or destination vertexes. Figure 3 shows an example of storage of graph in Figure 1. In this example, edges are stored according to their labels (i.e., a, b, or c), and there are two versions for each label; i.e., sorted according to the source (`*_src`) or destination (`*_dst`) vertexes.

4.3 Processing on FPGA

4.3.1 Join and Sort Modules

Our basic strategy to process an RPQ using an FPGA is to take advantage of the pipeline parallelism as much as possible. To this end, we consider that a (simple) RPQ is a multi-way join over edges where each join input corresponds to one of the path step in the RPQ. More precisely, we use *sort-merge join* to implement this pipeline, because sort is suitable for FPGAs and can be executed efficiently [15].

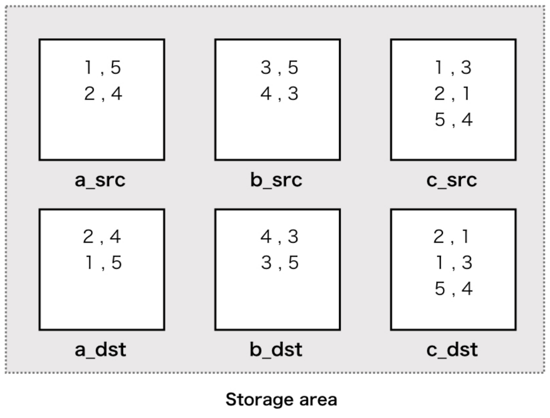
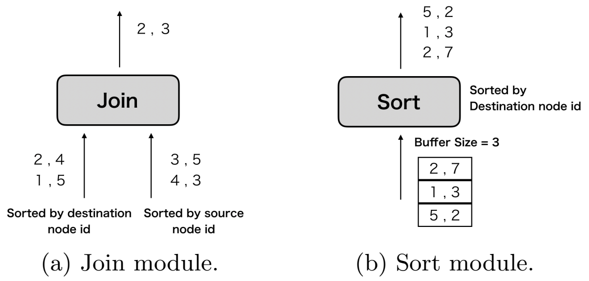


Figure 3: An overview of the proposed scheme

For this reason, we implemented two modules, namely, *join* and *sort* modules as the building blocks of our proposed scheme. Figure 4 depicts a simple illustration of the modules.

The join module takes as input two list of edges and outputs the composed paths if two edges being joined are connected; i.e., the destination of left input is equal to the source of right input. In this example, edges $(2, 4)$ and $(4, 3)$ are joined and $(2, 3)$ is output as the result. To perform this efficiently, we assume that left (right) input is sorted according to the destination (source) vertexes of the edges, thereby making it possible to perform sort-merge join in the module.

Regarding the sort module, we implemented the merge sort optimized for the FPGA implementation with reference to the algorithm described in [15]. This module is necessary, because the output of join module is not sorted; if we want to construct a pipeline, the input for subsequent join module requires the input is sorted according to source/destination vertexes. In this work, since we construct left-deep join tree, the sort module sorts the input according to the destination vertexes.



(a) Join module. (b) Sort module.

Figure 4: Modules implemented on FPGA.

4.3.2 Processing Pipeline on FPGA

This section shows how to process RPQs on an FPGA using the the modules described in Section 4.3.1. As already mentioned above, our basic strategy is to construct left-deep join tree using the join and sort modules. For example, let us take an RPQ $R = a \circ b \circ b \circ c$, we construct a processing pipeline as shown in Figure 5.

In the figure, `a_dst`, `b_src` and `c_src` are transferred from the host to the device and fed into the corresponding join modules. More precisely, the host first writes these data onto the global memory (off-chip DRAM) on the FPGA board. Then each modules can read necessary data from the global memory. The process proceeds in a staged manner; i.e., as soon as the system activates the first join module, the results are passed to the next sort module in a streaming manner, and the sort module stores the received data in its (sort) buffer. When the buffer becomes full or detects the end of the join results, it sorts the data in the buffer and pass them to the join module in the next stage. Notice that sort is a blocking operator. In other words, a sort can start only when all input is ready. For this reason, the intermediate results needs to be pooled in a buffer. The problem occurs when the buffer is overflowed due to an excessive size of intermediate results. In such cases, the intermediate results in the current (filled) buffer is first sent to the upper join module, and the remaining intermediate results are sent to the upper join module afterwords. In fact, the join module needs to perform join between left input (from sort buffer) and right input (from the host) for two or more times, and the right input needs to be fully read repeatedly, which will affect the performance.

Note that, in the proposed method, we first create on the FPGA a processing pipeline with predefined length. Given an RPQ, if its length is shorter than the length of pipeline, we can process the query without any reconfiguration. Let consider the example of pipeline in the Figure 5 consisting of five stages. When RPQ query with length 4 is given, we can process it with the pipeline without reconfiguration by 1) the first to fourth stages process as explained above and 2) the join and sort module at the fifth stage just pass the received data to the next module without any modification.

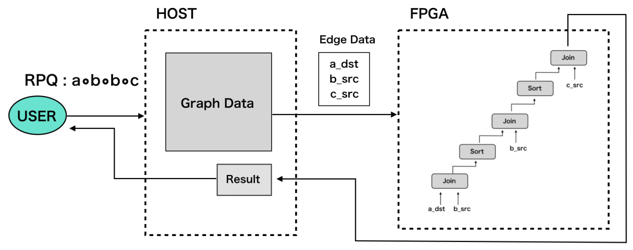


Figure 5: Process overview.

4.3.3 Parallelized Pipelining

To make the best use of the FPGA's performance, it is quite important to improve the parallelism. When considering the above mentioned pipeline configuration, as can easily be observed, the modules in higher stages tend to be inactive for long period of time until the left input is ready. Such a situation becomes even worse when processing long RPQs, because left inputs are processed stage-wisely.

In order to alleviate this problem, we propose a parallel pipeline configuration as shown in Figure 6, where a process pipeline is divided into two parallel pipelines which are executed in parallel. The example is based on a query

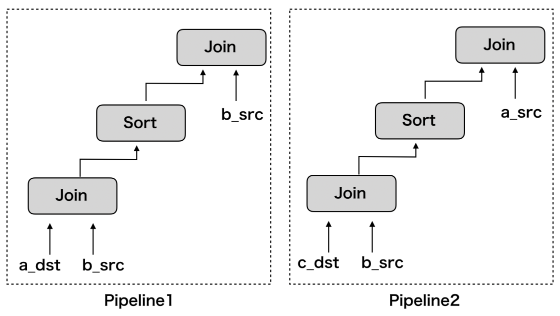


Figure 6: Configuration using two processing pipelines

$R = a \circ b \circ b \circ c \circ b \circ a$. In the proposed configuration, we divide the pipeline into two parts, namely, the first half ($R_1 = a \circ b \circ b$) and second half ($R_2 = c \circ b \circ a$). These partial RPQs are processed in parallel on the FPGA.

Notice that the results from these pipelines are not complete, because they are based on partial RPQs. To get the final (correct) results, we need the last join between the results corresponding to R_1 and R_2. Let us assume that this final join is performed on the FPGA, the following process would be needed: 1) the results from two (partial) pipelines are temporarily stored in the global memory of the FPGA; and 2) when the partial results are ready, we read the results from the memory and perform the final join. One of the major drawbacks of this approach is that the access cost to global memory is expensive, which will significantly affect the performance. Another drawback is that the size of intermediate results may not fit in the global memory.

Those having observed, in our proposed method, we decided to perform the final join on the host side using hash join. Figure 7 shows the overview. In the sequel discussion, we call the basic serial pipeline configuration *serial configuration*, while the improved parallel pipeline configuration *parallel configuration*.

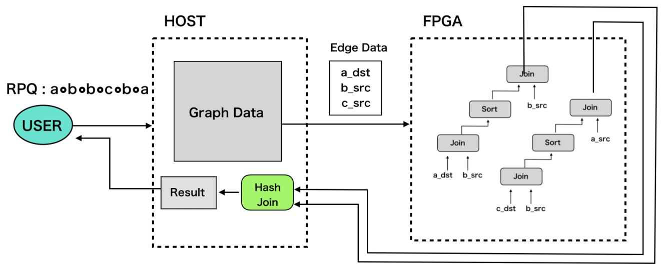


Figure 7: An overview of parallel configuration.

5. PERFORMANCE EVALUATION

5.1 Experimental Setup

To test the performance of the proposed scheme, we have conducted several experiments. The experimental environment is shown in Table 2.

5.2 Dataset

Table 2: Experimental environment.

CPU	Intel Core i7-7700K 3.60 GHz × 8
OS	Linux version 3.10.0
Memory	31.1GiB
Compiler	gcc-4.8.5
FPGA	Xilinx Kintex UltraScale FPGA KCU1500
RDBMS	PostgreSQL 9.2.24

In this experiment, we used two graph datasets, namely, advogato [17] and DBLP-Citation-network V10 [24]. Table 3 shows information such as the number of vertexes, and the details are described below.

Table 3: datasets

dataset	#vertexes	#edges	#label types
advogato	6,541	51,127	3
DBLP	4,850,632	38,973,022	6

5.2.1 *Advogato Dataset*

Advogato is a directed graph created from an on-line community platform for free software developers. Each edge is labeled with the trust between the users at the both ends of the edge.

5.2.2 *DBLP-Citation-network V10 Dataset*

DBLP-Citation-network is a dataset that contains about 3 million papers extracted from DBLP bibliography. Each paper contains paper ID, title, authors, cited paper IDs, venue and so on. In this experiment, we extracted paper ID, authors, cited paper IDs, and venue, and generated a heterogeneous graph consisting of different types of vertexes, i.e., paper (P), author (A) and venue (V), and edges labeled either $A \xrightarrow{writing} P$, $P \xrightarrow{written\ by} A$, $V \xrightarrow{publishing} P$, $P \xrightarrow{published\ in} V$, $P \xrightarrow{citing} P$ and $P \xrightarrow{cited\ by} P$.

5.3 Performance on advogato dataset

In the first experiment, we compare the performance of the proposed scheme with Path Index 1, an index-based method for RPQs which is known to be the fastest. The parameter k, which controls the length of indexed paths, is set to be $1 \leq k \leq 3$; i.e., simple RPQs of length up to three can be answered by just looking up the indexes.

We measured the execution time of RPQs with different length using different configurations, i.e., serial and parallel configuration. More precisely, we randomly selected RPQs of the same length, and computed the average of the execution times of 10 trials. Note that, for the proposed scheme, RPQs with the length shorter than three was processed using serial configuration only, because it is not possible to divide the pipeline into two. Besides, the sort buffer size for each sort module was 8,192 edges.

The results are shown in Figure 8. We can observe that, with the serial configuration, the proposed method was slower than Path Index when $|R| = 2$. However, when $R| \geq 3$, the proposed scheme is about 1.72x to 2.31x faster. Furthermore, in the parallel configration ($|R| = 4, 5$), the proposed method is about 15.3x to 23.6x faster than Path Index.

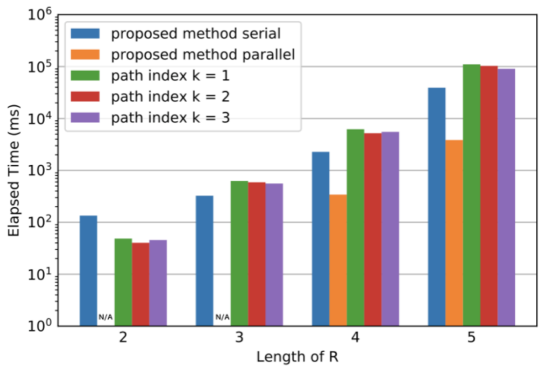


Figure 8: Performance with advogato dataset.

5.4 Performance with DBLP-Citation-network V10 Dataset

In this experiment, we conducted experiments using DBLP-Citation-network V10 dataset, which is larger than advogato dataset.

As for experimental query generation, we cannot generate queries in a completely random way due to the fact that there are some dependencies between edge labels and different types of vertexes; e.g., *writing* ∘ *publishing* never appears. For this reason, we chose five typical queries:

q_1: *writing* ∘ *written by*

q_2: *writing* ∘ *citing* ∘ *written by*

q_3: *writing* ∘ *published in* ∘ *publishing* ∘ *written by*

q_4: *publishing* ∘ *written by* ∘ *writing* ∘ *published in*

q_5: *writing* ∘ *citing* ∘ *published in* ∘
publishing ∘ *citing* ∘ *written by*

Besides, due to the excessive size of query results, it is not feasible to enumerate all occurrences of query results. Instead, we fixed the starting vertex[1], and find all destination vertexes that are reachable via the query RPQ being processed.

For the comparative method, we could not generate Path Index because of the size of index became too big (more than 10 GB for $k = 2$). For this reason, we stored the graph in a relational database (PostgreSQL) using a table with schema in Table 4. When processing a query, we convert the RPQ into an SQL; e.g., q_1 can be converted into the following SQL and executed on the database.

```
SELECT e1.src, e2.dst
FROM edges AS e1, edges AS e2
WHERE e1.src = 260069 -- This specifies
                         the starting vertex.
AND e1.label_id = 0 AND e2.label_id = 1
AND e1.dst = e2.src;
```

In this experiment, the size of sort buffer was 32,768 edges.

The results are shown in Figure 9. When using serial configuration, the performance of the proposed method was worse than the comparative method. By contrast, with the

[1]In the experiments, we used author "Hiroyuki Kitagawa" or venue "very large data bases" as the starting vertex.

Table 4: Schema structure of the table in PostgreSQL

(a) edges

src	dst	label_id
163954	1766548	0
1766548	163954	1
⋮	⋮	⋮

(b) labels

label_id	label
0	writing
1	written by
2	publishing
3	published in
4	citing
5	cited by

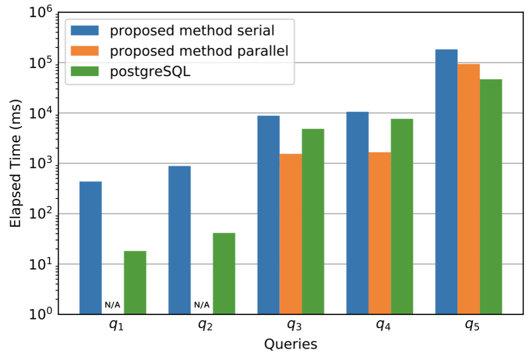


Figure 9: Performance for large graph(DBLP-Citation-network V10

parallel configuration, the proposed method was about 3.14x to 4.61x faster than the comparative method with q_3 and q_4. However, for q_5, the proposed scheme took longer execution time than PostgreSQL. We will discuss this later, but this is due to the restriction of the current implementation on FPGA.

5.5 Performance with Varying Buffer Size

In this experiment, we investigated the effect of buffer size in sort modules using serial configuration. We tested RPQs of length more than 3, because sort module is not used when processing RPQ of length 2. We used five buffer sizes, namely, 2,048, 4,096, 8,192, and 16,384 and 32,768.

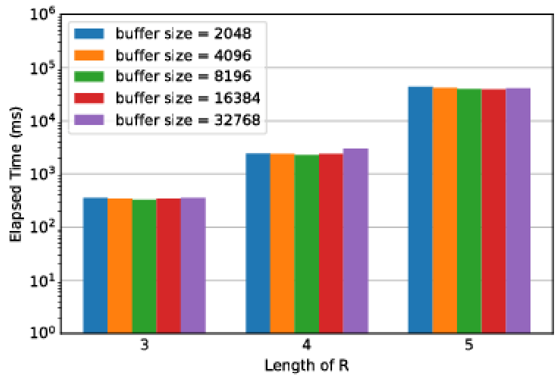


Figure 10: Performance changes with buffer size(advogato)

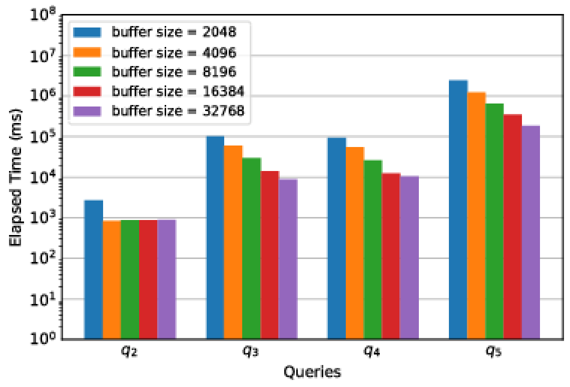


Figure 11: Performance changes with buffer size(DBLP-Citation-network V10)

The results are shown in Figures 10 and 11. From the result, we can observe that the buffer size has little influence on the performance when the data size is small (advogato). However, when processing large dataset (DBLP-Citation-network V10), the buffer size has a significant impact on the performance.

5.6 Resource Consumption

Table 5 shows the utilization of FPGA resources in the proposed method. Different rows for sort module represent the resource usage rate with different buffer sizes (the numbers in parentheses next to sortmodule indicate the buffer size). Besides, the lowest row ("Others") indicates the resource usage rate for the stuffs other than join or sort module, e.g., I/O modules from/to global memory, etc. As we can see from the table, as for LUT and Register, the resources required by each module are less than 1%. On the other hand, in terms of Block RAM (BRAM) usage, if we increae the buffer size of sort module, the BRAM resource uasage increases, because buffer is implemented using BRAM.

5.7 Discussion

In the proposed method, we can see that the performance is strongly affected by the datasets, which differ in size, label distribution, etc. More precisely, when processing small dataset (advogato), the performance of the proposed scheme was better than Path Index even with serial configuration. On the other hand, when processing large dataset (DBLP-Citation-network V10), the performance of the proposed scheme is significantly worse than the comparative method (PostgreSQL). This is due to the fact that the number of intermediate results at each join stage increases. In fact, join modules at the second or higher stages receives the left input from the sort module of the previous stage. As is explained in Section 4.3.2, when the intermediate join result overflows the sort buffer, the sort module send the buffer to the subsequent join module. Then, the join module process the left input by performing sort-merge join, which requires full scan of the right input (containing edges with a specific label sorted according to the source vertexes). This process repeats until the sort buffer becomes empty. Obviously, the join module needs to read entire right input repeatedly, which deteriorates the performnace. In summary,

Table 5: FPGA resource usage

	LUT	Register	Block RAM
Join Module	4187 (0.7%)	4613 (0.4%)	24 (1.2%)
Sort Module (2048)	1626 (0.3%)	1694 (0.1%)	8 (0.4%)
Sort Module (4096)	1673 (0.3%)	1709 (0.1%)	15 (0.7%)
Sort Module (8192)	1680 (0.3%)	1723 (0.1%)	32 (1.6%)
Sort Module (16384)	1730 (0.3%)	1735 (0.1%)	64 (3.2%)
Sort Module (32768)	1905 (0.3%)	1752 (0.1%)	128 (6.3%)
Others	4311 (0.8%)	4851 (0.4%)	0 (0.0%)

the performance of the proposed scheme primarily depends on whether the size of intermediate results fits in the sort buffers. This can be confirmed by the experimental results in Section 5.5. When processing large dataset, the frequency of buffer overflow at sort modules decreases when using large sort buffers. Coping with this problem is a part of our future work.

6. CONCLUSION

In this paper, we have proposed an RPQ accelerator using an FPGA. We have used two types of modules, join and sort modules, and constructed processing pipeline by combining the modules. To further exploit the parallelism, we have also proposed the parallel configuration where RPQ pipeline is divided into two and processed in parallel. We have tested the performance of the proposed scheme using two datasets, and the experimental results showd that 1) when the dataset is small, the proposed scheme is up to 23.6x faster then Path Index. The current issue is that the performance degrades as the dataset gets larger. In the future, we plan to develop a method to address such issues. In addition, we plan to extend the scheme to support more complecated RPQs containing operators other than composition, such as recursion, negation, etc.

7. ACKNOWLEDGEMENT

This research was partly supported by the cooperative research with Sky Co. (CPE01017). We would like to also thank to Prof. Ryohei Kobayashi at the Center for Computational Sciences, University of Tsukuba for his useful comments to our work.

8. REFERENCES

[1] IEEE P1076 Working GroupVHDL Analysis and Standardization Group (VASG) . `http://www.eda-twiki.org/cgi-bin/view.cgi/P1076/WebHome`.

[2] Introducing the next step in Netezza ' s evolution: IBM Integrated Analytics System. `https://www.ibm.com/us-en/marketplace/puredata-system-for-analytics`.

[3] OpenCL Reference Pages. `https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/`.

[4] openCypher. `http://www.opencypher.org/`.

[5] SPARQL 1.1 Query Language. `https://www.w3.org/TR/sparql11-query/`.

[6] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, April 2006.

[7] D. Agarwal and B.-C. Chen. Machine learning for large scale recommender systems, 2011.

[8] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[9] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *Design & Test of Computers, IEEE*, 26:8 – 17, 09 2009.

[10] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A Graphical Query Language Supporting Recursion. *SIGMOD Rec.*, 16(3):323–330, Dec. 1987.

[11] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 217–226, New York, NY, USA, 2017. ACM.

[12] G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France*, pages 636–639, March 2016.

[13] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[15] R. Kastner, J. Matai, and S. Neuendorffer. Parallel programming for fpgas. *CoRR*, abs/1805.03648, 2018.

[16] A. "Koschmieder and U. Leser. Regular path queries on large graphs. In A. Ailamaki and S. Bowers, editors, *Scientific and Statistical Database Management*, pages 177–194, Berlin, Heidelberg, 2012.

Springer Berlin Heidelberg.

[17] P. Massa, M. Salvetti, and D. Tomasoni. Bowling alone and trust decline in social network sites. In *In Proc. Int. Conf. Dependable, Autonomic and Secure Computing*, pages 658–663, 2009.

[18] R. Mehra and R. Verma. Area efficient fpga implementation of sobel edge detector for image processing applications.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[20] C. Schmitt, M. Schmid, F. Hannig, J. Teich, S. Kuckuk, and H. Kstler. Generation of multigrid-based numerical solvers for fpga accelerators.

[21] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415, New York, NY, USA, 2017. ACM.

[22] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. In *In VLDB' 11*, 2011.

[23] Y. Sun, B. Norick, J. Han, X. Yan, P. S. Yu, and X. Yu. Pathselclus: Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. *ACM Trans. Knowl. Discov. Data*, 7(3):11:1–11:23, Sept. 2013.

[24] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: Extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 990–998, New York, NY, USA, 2008. ACM.

[25] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM.

[26] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.