A New CPU-FPGA Heterogeneous gStore System

Xunbin Su, Yinnian Lin, and Lei Zou(✉)

Peking University, Beijing, China
{suxunbin,linyinnian,zoulei}@pku.edu.cn

Abstract. In this demonstration, we present a new CPU-FPGA heterogeneous gStore system. The previous gStore system is based on CPU and has low join query performance when the data size is too big. We implement a FPGA-based join module to speed up join queries. Furthermore, we design a FPGA-friendly data structure called FFCSR to facilitate it. We compare our new system with the previous one on the LUBM2B dataset. Experimental results demonstrate that the new CPU-FPGA heterogeneous system performs better than the previous one based on CPU.

Keywords: gStore · FPGA · FFCSR · Join query acceleration

1 Introduction

Recently in large-scale data retrieval and query, graph database has been applied in a wide range of fields. gStore [3] is a graph-based RDF data management system on multi-core CPU that supports SPARQL 1.1 standard query language defined by W3C. It stores RDF data as a labeled directed graph G (called RDF graph), where each vertex represents a subject or an object and each edge represents a predicate. Given a SPARQL query, we can also represent it as a query graph Q. Then gStore employs subgraph matching of Q over G to get query results. Specific technical details of gStore have been published before [6] and [7].

Though gStore can already support the storage and query of billions of RDF triples, the query performance significantly decreases as the data size increases, especially join queries (SPARQL queries involving join process). Two main performance bottlenecks for join queries are a large number of list intersections and too many reading Key-Value store operations from the external storage.

To overcome the two problems above and improve the overall performance of gStore, we introduce the field-programmable gate array (FPGA). A FPGA has many advantages including low energy consumption, high concurrency, programmability and simpler design cycles. In recent years, not only has GPU been widely used in graph computation [1] but also using FPGAs for hardware acceleration has gradually become a trend. For example, FPGAs are often used for accelerating some common graph algorithms like breadth-first search (BFS) and single-source shortest path (SSSP) [5].

X. Wang et al. (Eds.): APWeb-WAIM 2020, LNCS 12318, pp. 517–522, 2020.
https://doi.org/10.1007/978-3-030-60290-1_39

In this demo, we design and implement a new CPU-FPGA heterogeneous gStore system that has a FPGA-based join module and a FPGA-friendly data structure called FFCSR, which can accelerate join queries and improve the overall performance of gStore.

We conduct a comparative experiment on the previous and the new version of gStore on LUBM2B dataset. Experimental results demonstrate that the new CPU-FPGA heterogeneous gStore system performs better compared with the previous one on CPUs.

2 System Overview

The whole gStore system consists of an offline part and an online part. The offline part consists of a build module and a load module, for data preprocessing. The online part consists of a SPARQL parser, a filter module and a join module, handling a SPARQL query. We depict the system architecture in Fig. 1.

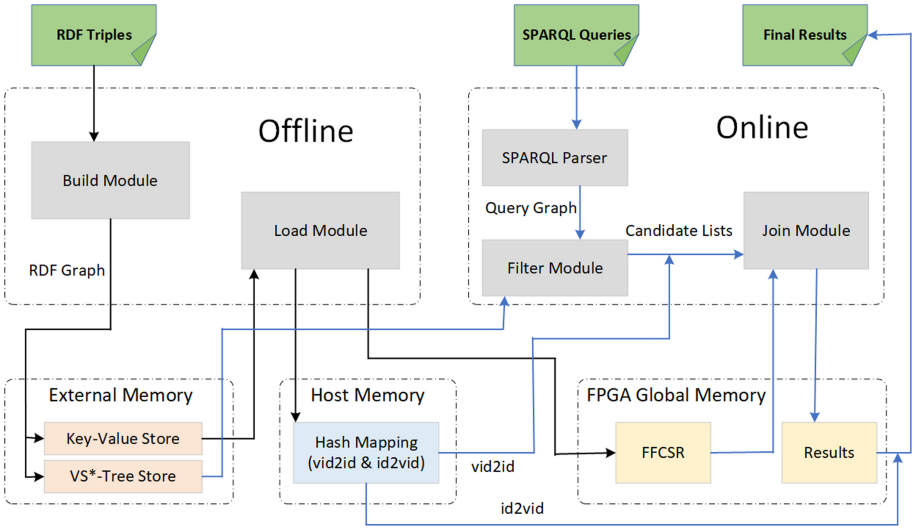


Fig. 1. The system architecture

Build Module. In the offline phase, gStore first accepts an RDF file and converts it to RDF triples. Then the build module uses adjacency list representation to build a RDF graph. The RDF graph is stored in external memory in the form of a Key-Value store and a VS*-tree store.

Load Module. GSI [4] proposes a GPU-friendly data structure to represent an edge-labeled graph, called PCSR. Inspired by [4], we build a FPGA-friendly Compressed Sparse Row during the load module, which we call FFCSR. Additionally, two sets of hash mappings are created to facilitate FPGA random access

to the FFCSR. We describe them in Fig. 2. Hash mappings are stored in host memory and the FFCSR is transferred to the FPGA's global memory (DRAM).

There are three components in the FFCSR: an index list, k offset lists and k adjacency lists, where k is the number of DDRs in DRAM. The index list has p triples, where p is the number of predicates. The first element of the i-th triple represents a DDR number. The second and the third are an out-edge offset initial position and an in-edge offset initial position. The i-th offset list and adjacency list are stored on the i-th DDR, which could be favorable for FPGA reading operations in parallel. The mapping *vid2id* converts the discrete but ordered vertex ids (*vid*) to the consecutive offset ids (*oid*) and the mapping *id2vid* is just opposite.

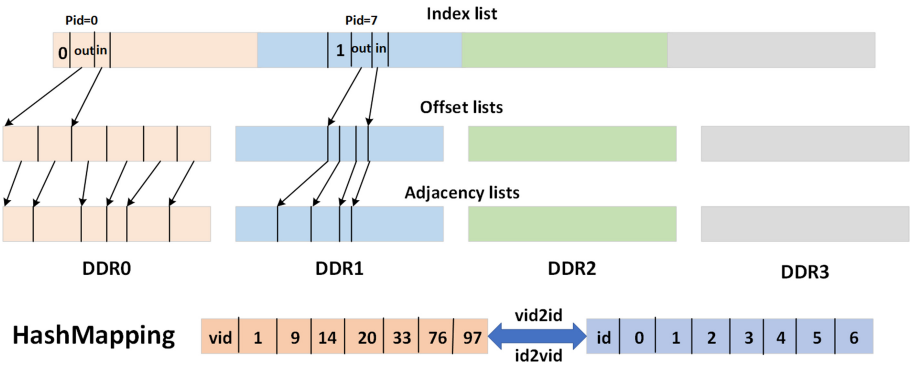


Fig. 2. The structure of FFCSR and corresponding hash mappings

SPARQL Parser. In the online phase, the SPARQL parser converts a SPARQL query given by a user to a syntax tree and then builds a query graph based on the syntax tree. The query graph is encoded into a signature graph with the similar encoding strategy that encodes RDF graphs.

Filter Module. After the SPARQL parser, the filter module uses the VS*-tree store to generate an ordered candidate list for each query node. This step is necessary to reduce the size of the input to the join module.

Join Module. The join module can be divided into two parts: a CPU host and a set of FPGA kernels. We adopt multi-step join [2] in the join module. The order which query node should be added in each step is determined by its priority. The priority of each query node is inversely proportional to the size of its candidate list because the smaller size leads to fewer intermediate results and a faster join. Then the host generates two types of candidate lists for FPGA kernels. One is mapped using the mapping *vid2id*, which we call CLR (candidate list for reading). The other is converted to a bitstring where the i-th bit represents whether $vid = i$ exists, which we call CLS (candidate list for searching). Finally,

FPGA kernels perform specific computational tasks to get results. Note that results should be restored using the mapping *id2vid*.

3 FPGA Kernel

A set of FPGA kernels are designed for computation of intersection among multiple candidate lists. They are controlled by the host. A FPGA kernel is designed as a 4-stage pipeline consisting of reading FFCSR and CLR, intersection computing, searching in CLS and writing back, as shown in Fig. 3.

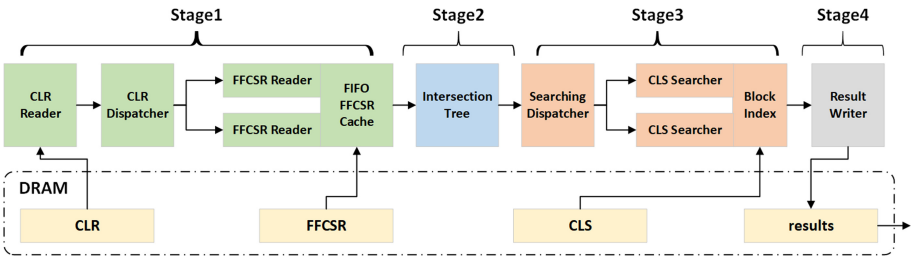


Fig. 3. The pipeline overview of a FPGA kernel

Reading FFCSR and CLR. A CLR reader reads one set of k elements at a time and passes them to a dispatcher for FFCSR reading. A dispatcher decides to assign tasks to FFCSR reading threads according to the *oid*. As soon as a thread gets an *oid*, it accesses the FFCSR in DRAM for corresponding adjacency list and stores it in a buffer in BRAM for the next stage.

Intersection Computing. We implement a merge-based intersection tree whose leaf nodes correspond to original adjacency lists, to find the common elements among all k adjacency lists in a bottom-up manner. At every cycle an intersection is conducted between each 2 sibling nodes. Because the BRAM is faster than the DRAM, we design a simple FIFO cache shared by the threads. Before a thread accesses the DRAM, it first searches in the cache in BRAM to see if the needed list is already cached to avoid unnecessary IO.

Searching in CLS and Writing Back. The *vids* generated by the intersection tree are sent to another dispatcher. This dispatcher conducts Round-Robin Scheduling and allocates a *vid* to a searching thread each cycle. Then each thread searches a given *vid* in the CLS. If the corresponding bit of the CLS is equal to one, the result will be written to the output buffer.

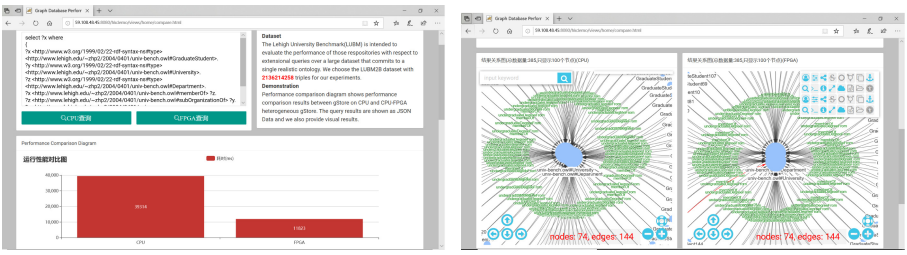
Optimization. We design a smaller in-BRAM block index for searching to reduce DRAM access. Note that in most cases the CLS is very sparse with many continuous zeros. Therefore, we use a bitstring index where one bit represents a fixed-length block of the CLS. The size of the block index is small enough to

be put in BRAM. A searching thread first checks the block index to determine whether a given *vid* is valid. The CLS in DRAM is searched only when the *vid* is valid.

4 Demonstration

In this demo, we use the famous Lehigh University Benchmark (LUBM) for evaluating the efficiency of our new gStore system. We choose the LUBM2B dataset with 2136214258 triples for our experiments. The host system is a supermicro 7046GT-TRF (Intel Xeon E5-2620 v4 2.10 GHz, 256 GB memory) which is equipped with a Xilinx alevo-u200 FPGA board (64GB global memory, 35 MB BRAM) via PCIe 3.0 with 16 lanes. The design on the FPGA is clocked with 330 MHz.

In Fig. 4(a), we input a join query and execute it with gStore on CPU and CPU-FPGA heterogeneous gStore, respectively. The performance comparison histogram is below to validate the effectiveness. In Fig. 4(b), the visual query results are shown to validate the correctness. Experimental results demonstrate that CPU-FPGA heterogeneous gStore can achieve up to 3.3× speedup for join queries.



(a) Query Page (b) Visual Results

Fig. 4. Demonstration of gStore

Acknowledgment. This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504 and NSFC under grant 61932001.

References

1. Alam, M., Perumalla, K.S., Sanders, P.: Novel parallel algorithms for fast multi-GPU-based generation of massive scale-free networks. Data Sci. Eng. **4**(1), 61–75 (2019). https://doi.org/10.1007/s41019-019-0088-6
2. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms. J. ACM (JACM) **65**(3), 16 (2018)

3. Shen, X., et al.: A graph-based RDF triple store. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 1508–1511. IEEE (2015)
4. Zeng, L., Zou, L., Özsu, M.T., Hu, L., Zhang, F.: GSI: GPU-friendly subgraph isomorphism. arXiv preprint arXiv:1906.03420 (2019)
5. Zhou, S., Prasanna, V.K.: Accelerating graph analytics on CPU-FPGA heterogeneous platform. In: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 137–144. IEEE (2017)
6. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: answering SPARQL queries via subgraph matching. Proc. VLDB Endow. **4**(8), 482–493 (2011)
7. Zou, L., Özsu, M.T., Chen, L., Shen, X., Huang, R., Zhao, D.: gStore: a graph-based SPARQL query engine. VLDB J.-Int. J. Very Large Data Bases **23**(4), 565–590 (2014)