Context-Free Path Querying Can be Fast if Prepared Properly

Arseniy Terekhov
!!!
Saint Petersburg State University
St. Petersburg, Russia

Artyom Khoroshev
!!!
!!!
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
semen.grigorev@jetbrains.com
Saint Petersburg State University
St. Petersburg, Russia
JetBrains Research
St. Petersburg, Russia

ABSTRACT

Recently proposed matrix multiplication based algorithm for context-free path querying (CFPQ) offloads the most performance-critical parts onto boolean matrices multiplication. Thus, it is possible to achieve high performance of CFPQ by means of modern parallel hardware and software. In this paper, we provide results of empirical performance comparison of different implementations of this algorithm on both real-world data and synthetic data for the worst cases.

KEYWORDS

Context-free path querying, transitive closure, graph databases, context-free grammar, GPGPU, CUDA, boolean matrix, matrix multiplication

1 INTRODUCTION

Language-constrained path querying [?], and particularly Context-Free Path Querying (CFPQ) [?], allows one to use formal grammars as constraints for paths: concatenation of the labels along the path is treated as a word, and a constraint on the path is a specification of the language which should contain specific words. CFPQ is widely used for graph-structured data analysis in such domains as biological data analysis, RDF, network analysis. Huge amount of the real-world data makes performance of CFPQ critical for practical tasks. Several algorithms for CFPQ based on such parsing techniques as (G)LL, (G)LR, and CYK are proposed recently [? ? ? ? ? ?].

One of the most promising algorithms is a matrix-based algorithm, proposed by Rustam Azimov [?]. This algorithm offloads the most critical computations onto boolean matrices multiplication. As a result, it is easy to implement, and allows one to utilize modern massive-parallel hardware for CFPQ. The implementation provided by the authors utilizes GPGPU by using cuSPARSE[1] library which is a floating point sparse matrices multiplication library. Although it does not use advanced algorithms for boolean matrices, it outperforms existing algorithms.

It is necessary to investigate the effect of the specific algorithms and implementation techniques on the performance of CFPQ. One problem is that no publicly available standard dataset for CFPQ algorithms evaluation which includes both graph-structured data and queries exists.

In this work, we do an empirical performance comparison of several implementations of matrices multiplication based algorithm for CFPQ on both real-world data and synthetic data for the worst cases. We make the following contributions in this paper.

(1) We provide a number of implementations of the matrix multiplication based CFPQ algorithm, which utilizes different modern software and hardware. The source code is available on GitHub: https://github.com/SokolovYaroslav/CFPQ-on-GPGPU
(2) We present a dataset which contains both real data and synthetic data for worst cases. This dataset contains data and queries in the simple textual format, so it can be easily used to evaluate other algorithms. The dataset is available on GitHub via the same link, and we hope that this dataset can form a base of the unified benchmark for CFPQ algorithms.
(3) We provide evaluation which shows that GPGPU utilization for CFPQ can significantly improve performance, and that there are still many open questions in this area.

2 MATRIX-BASED ALGORITHM FOR CFPQ

Matrix-based algorithm for CFPQ was proposed by Rustam Azimov [?]. This algorithm can be expressed in terms of operations over matrices (see listing 1), and it is a sufficient advantage for implementation. It was shown that the utilization of GPGPU improves the context-free path querying performance significantly in comparison to other implementations [?] even if float matrices are used instead of boolean matrices.

Listing 1 Context-free path quering algorithm

```
function CONTEXTFREEPATHQUERYING(D, G)
    n ← the number of nodes in D
    E ← the directed edge-relation from D
    P ← the set of production rules in G
    T ← the matrix n × n in which each element is ∅
    for all (i, x, j) ∈ E do                    ▷ Matrix initialization
        T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
    while matrix T is changing do
        T ← T ∪ (T × T)                   ▷ Transitive closure calculation
    return T
```

Here $D = (V, E)$ is the input graph and $G = (N, \Sigma, P)$ is the input grammar. Each cell of the matrix T contains the set of nonterminals such that $N_k \in T[i, j] \iff \exists p = v_i \ldots v_j$—path in D, such that $N_k \overset{*}{\underset{G}{\Rightarrow}} \omega(p)$, where $\omega(p)$ is a word formed by the labels along the path p. Thus, this algorithm solves reachability problem, or, according to Hellings [?], processes CFPQs by using relational query semantics.

[1] cuSparse is a library for sparse matrices multiplication on GPGPU. Official documentation: https://docs.nvidia.com/cuda/cusparse/index.html. Access date: 12.03.2019

The performance-critical part of the algorithm is matrix multiplication. Note, that the set of nonterminals is finite, and we can represent the matrix T as a set of boolean matrices: one for each nonterminal. In this case the operation of matrix update is $T_{N_i} \leftarrow T_{N_i} + (T_{N_j} \times T_{N_k})$ for each production $N_i \rightarrow N_j\ N_k$ in P. Thus we can reduce CFPQ to boolean matrices multiplication. After such transformation, we can apply the next optimization: we can skip update if the matrices T_{N_j} and T_{N_k} have not been changed at the previous iteration.

Thus, the most important part is the efficient implementation of operations over boolean matrices. In this paper, we compare the effects of different approaches to matrices multiplication. All our implementations are based on the optimized version of the algorithm.

3 IMPLEMENTATION

We implement the matrix-based algorithm for CFPQ by using several different programming languages and tools. We aim to investigate the effects of the following features of an implementation.

- **GPGPU utilization.** It is well-known that GPGPUs are suitable for matrices operations, but the performance of the whole solution depends on the implementation details. For example, overhead on data transferring may negate the effect of parallel computations. Can GPGPUs utilization for CFPQ improve performance in comparison with the CPU version?
- **Existing libraries utilization** is a good practice in software engineering. Is it possible to achieve higher performance by means of existing libraries for matrices operations or do we need to create our own solution to get more control?
- **Low-level programming**. GPGPU programming traditionally involves low-level programming in C-based languages (CUDA C, OpenCL C). But can we achieve a high-performance solution with high-level languages such as Python?
- **Sparse matrices.** Real graphs are often sparse. Can we gain more performance improvement by using sparse matrix representation for CFPQ?

We provide the following implementations for investigation.

- CPU-based solutions
  [Scipy] Sparse matrices multiplication by using Scipy [?] in Python programming language.
  [M4RI] Dense matrices multiplication by using m4ri[2] [?] library which implements the Method of Four Russians [?] in C language. This library is choosen because it is a performant implementation of the Method of Four Russians [?].
- GPGPU-based solutions
  [GPU4R] Our own implementation of the Method of Four Russians in CUDA C.
  [GPU_N] Our own implementation of the naïve boolean matrix multiplication in CUDA C with boolean values treated as bits and packed into `uint_32`.
  [GPU_Py] Manual implementation of naïve boolean matrix multiplication in Python by using numba compiler[3]. Boolean values are packed into `uint_32`.

As far as a set of matrices and its size can be calculated at the start of computations, all the GPGPU based implementations allocate all required memory on the GPGPU once, at the start of computations. This significantly reduces the overhead on data transferring: all input data is loaded to the GPGPU at the start, and the result is loaded from the GPGPU to the host at the end. As a result, there is no active data transferring and memory allocating during query computation.

4 DATASET DESCRIPTION

We created and published a dataset for CFPQ algorithms evaluation. This dataset contains both the real-world data and synthetic data for different specific cases, such as the theoretical worst case, or the worst cases specific to matrices representations.

Our goal is to evaluate querying algorithms, not graph storages or graph databases, so all data is presented in a text-based format to simplify usage in different environments. Grammars are in Chomsky Normal Form, and graphs are represented as a list of triples (edges). Some details of the data representation can be found in the Appendix.

The variants of the *same generation query* [?] are an important example of queries that are context-free but not regular, so we use this type of queries in our evaluation. The dataset includes data for the following cases. Each case is a pair of a set of graphs and a set of grammars: each query (grammar) should be applied to each graph.

[RDF] The set of the real-world RDF files (ontologies) from [?] and two variants of the same generation query which describes hierarchy analysis. The first query is the grammar G_4:

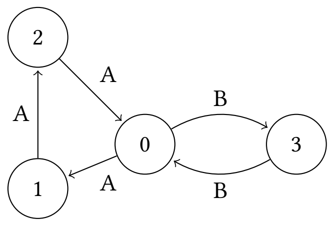
$$\begin{array}{ll} s \rightarrow SCOR\ s\ SCO & s \rightarrow TR\ s\ T \\ s \rightarrow SCOR\ SCO & s \rightarrow TR\ T \end{array}$$

The second one is the grammar G_5: $s \rightarrow SCOR\ s\ SCO \mid SCO$.

[Worst] The theoretical worst case for CFPQ time complexity proposed by Hellings [?]: the graph is two cycles of coprime lengths with a single common vertex. The first cycle is labeled by the open bracket and the second cycle is labeled by the close bracket. Query is a grammar for the $A^n B^n$ language. The example of such graph and grammar is presented in figure 1.

$$\begin{array}{l} s \rightarrow A\ s\ B \\ s \rightarrow A\ B \end{array}$$

(a) Grammar G_1 for $\{A^n B^n\}$



(b) Input graph D_1 for the worst case with `A` as the open bracket, and `B` as the close bracket

Figure 1: Graph and grammar for the worst case

[Full] The case when the input graph is sparse, but the result is a full graph. Such a case may be hard for sparse matrices representations. As an input graph, we use a cycle, all edges of

[2] Actually we use pull request which is not merged yet: https://bitbucket.org/malb/m4ri/pull-requests/9/extended-m4ri-to-multiplication-over-the/diff. The original library implements operations over $GF(2)$, and this pull request contains operations over boolean semiring

[3] Numba is a JIT compiler which supports GPGPU for a subset of Python programming. Official page: http://numba.pydata.org/. Access date: 03.05.2019

which are labeled by the same token. As a query we use two grammars which describe the sequence of tokens of arbitrary length: the simple ambiguous grammar G_2: $s \to s\ s \mid A$, and the highly ambiguous grammar G_3: $s \to s\ s\ s \mid A$.

[Sparse] Sparse graphs from [?] are generated by the GT-graph graph generator, and emulate realistic sparse data. Names of these graphs have the form Gn-p, where n corresponds to the total number of vertices, and p is the probability that some pair of vertices is connected. The query is the same generation query represented by the grammar G_1 (figure 1).

5 EVALUATION

We evaluate all the described implementations on all the datasets and the queries presented. We compare our implementations with [?]. We exclude the time required to load data from files. The time required for data transfer is included.

For evaluation, we use a PC with Ubuntu 18.04 installed. It has Intel core i7 8700k 3,7HGz CPU, DDR4 32 Gb RAM, and Geforce 1080Ti GPGPU with 11Gb RAM.

The results of the evaluation are summarized in the tables below. Time is measured in seconds unless specified otherwise. The result for each algorithm is averaged over 10 runs. The cell is left blank if the time limit is exceeded, or if there is not enough memory to allocate the data.

The results of the first dataset **[RDF]** are presented in table 1. We can see, that in this case the running time of all our implementations is smaller than of the reference implementation, and all implementations but **[CuSprs]** demonstrate similar performance. It is obvious that performance improvement in comparison with the first implementation is huge and it is necessary to extend the dataset with new RDFs of the significantly bigger size.

Table 2: Evaluation results for the worst case

#V	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
16	0.032	< 1	0.008	0.002	0.027	0.309
32	0.118	0.001	0.034	0.008	0.136	0.441
64	0.476	0.041	0.133	0.032	0.524	0.988
128	2.194	0.226	0.562	0.129	2.751	3.470
256	15.299	1.994	3.088	0.544	11.883	15.317
512	121.287	23.204	13.685	2.499	43.563	102.269
1024	1593.284	528.521	88.064	19.357	217.326	1122.055
2048	-	-	-	325.174	-	-

Results of the theoretical worst case (**[Worst]** datatset) are presented in table 2. This case is really hard to process: even for a graph of 1024 vertices, the query evaluation time is greater than 10 seconds even for the most performant implementation. We can see, that the running time grows too fast with the number of vertices.

Table 3: Sparse graphs querying results

Graph	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
G5k-0.001	10.352	0.647	0.113	0.041	0.216	5.729
G10k-0.001	37.286	2.395	0.435	0.215	1.331	35.937
G10k-0.01	97.607	1.455	0.273	0.138	0.763	47.525
G10k-0.1	601.182	1.050	0.223	0.114	0.859	395.393
G20k-0.001	150.774	11.025	1.842	1.274	6.180	-
G40k-0.001	-	97.841	11.663	8.393	37.821	-
G80k-0.001	-	1142.959	88.366	65.886	-	-

The next is the **[Sparse]** datatset presented in table 3. The evaluation shows that sparsity of graphs (value of parameter p) is important both for implementations which use sparse matrices and for implementations which use dense matrices. Note that the behavior of the sparse matrices based implementation is as expected, but for dense matrices we can see, that more sparse graphs are processed faster. Reasons for such behavior demand further investigation. Note that we estimate only the query execution time, so it is hard to compare our results with the results presented in [?]. Nevertheless, the running time of our **[GPU_N]** implementation is significantly smaller than the one provided in [?].

The last dataset is **[Full]**, and results are shown in table 4. As we expect, this case is very hard for sparse matrices based implementations: the running time grows too fast. This dataset also demonstrates the impact of the grammar size. Both queries specify the same constraints, but the grammar G_3 in CNF contains 2 times more rules then the grammar G_2, so, the running time for big graphs differs by more than twice.

Finally, we can conclude that GPGPU utilization for CFPQ can significantly improve performance, but more research on advanced optimization techniques should be done. On the other hand, the high-level implementation (**[GPU_Py]**) is comparable with other GPGPU-based implementations. So, it may be a balance between implementation complexity and performance. Highly optimized existing libraries can be of some use: the implementation based on m4ri is faster than the reference implementation and the other CPU-based implementation. Moreover, it is comparable with some GPGPU-based implementations in some cases. Sparse matrices utilization demands more thorough investigation. The main question is if we can create an efficient implementation for sparse boolean matrices multiplication.

6 DISCUSSION

Zeros in sparse matrices.

Overhead on matrices convertion and transferring when run on GPGPU.

No convertion when run on CPU. Beter then Neo4j and other BD with non-matrix representation of graphs.

7 CONCLUSION AND FUTURE WORK

We provide a number of implementations of the matrix-based algorithm for context-free path querying, collect a dataset for evaluation and provide results of evaluation of our implementation on the collected dataset. Our evaluation shows that GPGPU utilization for boolean matrices multiplication can significantly increase the performance of CFPQs evaluation, but requires more research of implementation details.

The first direction for future research is a more detailed investigation of the CFPQ algorithms. We should do more evaluation on sparse matrices on GPGPUs and investigate techniques for high-performance GPGPU code creation. Also, it is necessary to implement and evaluate solutions for graphs which do not fit in RAM, and for big queries which disallow to allocate all required matrices on a single GPGPU. We hope that it is possible to utilize existing techniques for huge matrices multiplication for this problem.

Another direction is the dataset improvement. First of all, it is necessary to collect more data, and more grammars/queries. It is of most importance to add more real-world graphs and more real-world queries to the dataset. Secondly, it is necessary to discuss

Table 1: RDFs querying results (time in milliseconds)

RDF			Query G_4						Query G_5					
Name	#V	#E	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
atm-prim	291	685	3	2	2	1	5	269	1	< 1	1	< 1	2	267
biomed	341	711	3	5	2	1	5	283	4	< 1	1	< 1	5	280
foaf	256	815	2	9	2	< 1	5	270	1	< 1	1	< 1	2	263
funding	778	1480	4	7	4	1	5	279	2	< 1	3	< 1	4	274
generations	129	351	3	3	2	< 1	5	273	1	< 1	1	< 1	2	263
people_pets	337	834	3	3	3	1	7	284	1	< 1	1	< 1	3	277
pizza	671	2604	6	8	3	1	6	292	2	< 1	2	< 1	5	278
skos	144	323	2	4	2	< 1	5	273	< 1	< 1	1	< 1	2	265
travel	131	397	3	5	2	< 1	6	268	1	< 1	1	< 1	3	271
unv-bnch	179	413	2	4	2	< 1	5	266	1	< 1	1	< 1	3	266
wine	733	2450	7	6	4	1	7	294	1	< 1	3	< 1	3	281

Table 4: Full querying results

#V	Query G_2						Query G_3					
	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs	Scipy	M4RI	GPU4R	GPU_N	GPU_Py	CuSprs
100	0.007	0.002	0.002	< 1	0.003	0.278	0.023	0.076	0.005	0.001	0.007	0.290
200	0.040	0.003	0.002	0.001	0.004	0.279	0.105	0.098	0.004	0.001	0.007	0.296
500	0.480	0.003	0.003	0.001	0.004	0.329	1.636	0.094	0.007	0.001	0.010	0.382
1000	3.741	0.007	0.005	0.001	0.006	0.571	13.071	0.106	0.009	0.001	0.009	0.839
2000	40.309	0.063	0.019	0.003	0.017	1.949	93.676	0.108	0.030	0.005	0.026	3.740
5000	651.343	0.366	0.125	0.038	0.150	99.651	1205.421	0.851	0.195	0.075	0.239	201.151
10000	-	1.932	0.552	0.315	0.840	1029.042	-	4.690	1.055	0.648	1.838	-
25000	-	33.236	7.252	5.314	15.521	-	-	70.823	15.240	10.961	36.495	-
50000	-	360.035	58.751	44.611	129.641	-	-	775.765	130.203	91.579	226.834	-
80000	-	1292.817	256.579	190.343	641.260	-	-	-	531.694	376.691	-	-

and fix the data format to be able to evaluate different algorithms. We believe that it is necessary to create a public dataset for CFPQ algorithms evaluation, and collaboration with the community is required.

ACKNOWLEDGMENTS

The research was supported by the Russian Science Foundation grant 18-11-00100 and a grant from JetBrains Research.