GRAM: A GPU-Based Property Graph Traversal and Query for HPC Rich Metadata Management

Wenke Li, Xuanhua Shi, Hong Huang, Peng Zhao, Hai Jin, Dong Dai, Yong Chen

GRAM: A GPU-based Property Graph Traversal and Query for HPC Rich Metadata Management

Wenke Li[1], Xuanhua Shi[1], Hong Huang[1], Peng Zhao[1], Hai Jin[1], Dong Dai[2], and Yong Chen[2]

[1] Services Computing Technology and System Lab
Big Data Technology and System Lab
Huazhong University of Science and Technology
Wuhan, 430074, China
[2] Department of Computer Science
Texas Tech University
Lubbock, Texas, 43104, US

Abstract. In HPC systems, rich metadata are defined to describe rich information about data files, like the executions that lead to the data files, the environment variables, and the parameters of all executions, etc. Recent studies have shown the feasibility of using property graph to model rich metadata and utilizing graph traversal to query rich metadata stored in the property graph. We propose to utilize GPU to process the rich metadata graphs. There are generally two challenges to utilize GPU for metadata graph query. First, there is no proper data representation for the metadata graph on GPU yet. Second, there is no optimization techniques specifically for metadata graph traversal on GPU neither. In order to tackle these challenges, we propose GRAM, a GPU-based property graph traversal and query framework. GRAM uses GPU to express metadata graph in *Compressed Sparse Row* (CSR) format, and uses *Structure of Arrays* (SoA) layout to store properties. In addition, we propose two new optimizations, parallel filtering and basic operations merging, to accelerate the metadata graph traversal. Our evaluation results show that GRAM can be effectively applied to user scenarios in HPC systems, and the performance of metadata management is greatly improved.

Keywords: Rich Metadata Management, Property Graph, Graph Traversal, GPU

1 Introduction

Graph structures are widely used in various domains to solve real problems, such as friend recommendation in social networks where people are vertices and their relationships are edges, or shortest path selection in digital maps where locations are vertices and routes connecting them are edges. Among all the graph structures, property graph [3] is one commonly used one, whose vertices and edges

are associated with arbitrary properties. Because of its richness in expressing the graph entities and their relationships, the property graph has been used widely in graph computing frameworks [13] and graph storage systems [1].

Recently, property graphs have been used in modeling metadata of large-scale parallel computing systems [7–9]. Unlike traditional metadata management [24] that relies on directory trees and *inode* data structure [22], property graph can utilize graph structure to represent and manage various entities and their complex relationships. This is particularly useful for the case where rich metadata like provenance is recorded and managed. In addition, using graph model, complex metadata queries can be easily expressed as graph traversal. To accomplish that, GraphTrek [7], an asynchronous graph traversal engine providing high access speed and supporting flexible queries, has been proposed and evaluated to show the effectiveness of property graph in managing rich metadata in HPC systems.

Because of the large volume of information contained in rich metadata, storing and querying them in property graph is still challenging. Although many property graph databases have been proposed and developed in recent years [2, 4,17,23], they have limitations regarding speed and throughput during managing rich metadata in performance critical usage scenarios, such as user audit [9] and provenance query [21,25]. In these two scenarios, efficient rich metadata querying is needed, which brings significant burden on modern CPUs, particularly in computation speed and memory bandwidth. Harish et al. [14] have found that many graph algorithms run faster on GPU, for example, the *Single Source Shortest Paths* (SSSP) algorithm and *Breadth-first search* (BFS) algorithm implemented on GPU can provide more than 100 times speedup. As we have described before, queries on graph-based rich metadata can be easily mapped to level-synchronous graph traversal operations, with extra filtering and path selection. This inspires us to cooperate GPU to further enhance the performance of rich metadata management.

It is non-trivial to use GPU to accelerate graph-based rich metadata management. On the model side, it lacks a proper metadata graph representation on GPU. On the algorithm side, the parallelism of graph traversal is largely limited by the super-step in level-synchronous traversal. In order to fully utilize the potentials of GPU, new optimization techniques are required for graph traversal.

To this end, we propose GRAM, a GPU-based property graph traversal and query framework for HPC rich metadata management. Our design focuses on reducing memory access overhead and improving procedure efficiency and utilization of GPU. Specifically, the amount of data processed by rich metadata graph traversal could be very large due to the attached arbitrary properties. Hence, GPU's high memory bandwidth helps significantly in reducing the memory access latency and improving the efficiency of memory access. Furthermore, since the data unit processed by property graph is independent, we can make full use of GPU's parallelism and storage resources to further improve the performance.

In GRAM, we arrange data using the *Structure of Arrays* (SoA) layout. Specifically, the graph topological data (vertices and their connecting edges) are represented using *Compressed Sparse Row* (CSR) which consists of three arrays; the property data attached to vertices and edges are put separately in other arrays. Through our property graph representation and layout, the metadata management activities are translated into arrays operations. The property graph traversal to query rich metadata is becoming a n-step iterative process. There are two array operations in each step: detecting whether one of the properties conforms to the filter criteria and gathering the vertices/edges with a qualified label. In GRAM, these two operations on arrays are optimized by GPU, while the complex relationships between the arrays are suitable for CPU to process. In addition, we use parallel filtering and basic operations merging to optimize the performance of GRAM. Our contributions in this study are three-fold:

- To the best of our knowledge, we are the first to utilize GPU for managing graph-based rich metadata generated in HPC systems.
- We propose metadata graph representation on GPU, combining CSR graph structure and SoA layout to represent graphs to represent and store rich metadata information.
- We parallelize filtering and merge basic operations in GRAM, and experimental results show that our design improves the performance of property graph traversal and query in metadata management usage scenarios.

The rest of the paper is organized as follows. The design and implementation details of GRAM are presented in Section 2, including overall architecture, metadata graph representation on GPU, metadata graph operations model on GPU, metadata operations translating and GRAM's optimizations. We evaluate the performance of GRAM, and present the results in Section 3. Related work is given in Section 4. Section 5 concludes the paper.

2 Design and Implementation

GRAM is designed to manage HPC rich metadata using GPU. In GRAM, the rich metadata graphs are stored in arrays and the queries on rich metadata, i.e., the graph traversal operations are mapped to GPU operations on these arrays. More design and implementation details will be discussed in this section. Specifically, Section 2.2 introduces rich metadata graph representation on GPU. After that, Section 2.3 states how rich metadata graph operations are modeled on GPU. In addition, Section 2.4 presents the translation details of rich metadata management and property graph traversal and query. Finally, Section 2.5 describes two optimizations proposed in GRAM to enhance the rich metadata query performance.

2.1 Overall Architecture

We design and implement GRAM, a GPU-based framework for HPC rich metadata management. GRAM is designed to support HPC rich metadata query

through property graph traversal on GPU. The overall architecture of GRAM is shown in Figure 1. It includes four modules internally: Query Interface, Metadata Translating, Query Engine, and Storage. Query Interface module receives user's metadata management requests, and forwards the requests to Query Engine module. Query Interface module interacts with Metadata Translating module through basic query operations. Metadata Translating module translates the representation of the property graph and maps metadata to Storage module. In this way, Query Engine can directly operate on arrays stored in the Storage module. Storage return results to Query Engine for further processing. These four parts work together to perform rich metadata graph traversal and query.

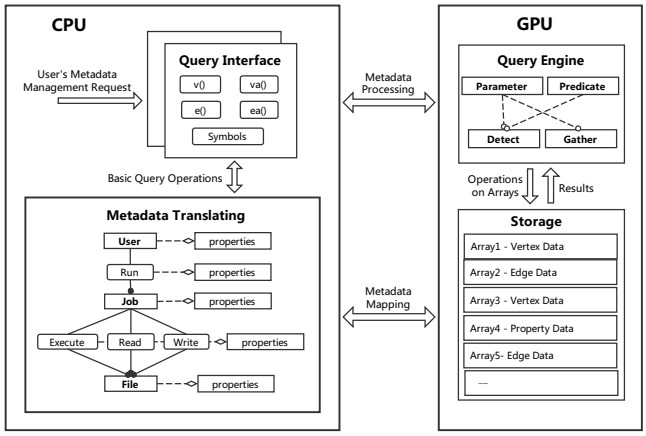


Fig. 1. Overall architecture of GRAM

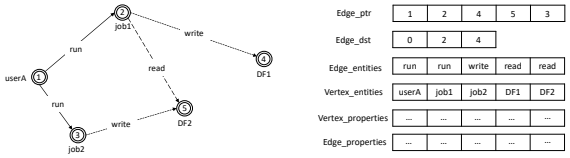
The four components are designed in both GPU and CPU. Query Interface and Metadata Translating module are expected to run on CPU. They preprocess the rich metadata graphs before executing any operation on GPU. Users submit their queries through a sequence of API calls to Query Interface module. Query Interface module works as a coordinator with necessary functional APIs to translates users' queries into a sequence of basic query operations. These basic query operations are then dispatched to the Metadata Translating module. The Metadata Translating module handles the relationships between the entities of the property graph. The Metadata Translating module and Query Interface module collaborate together to translate the sequence of queries into metadata operations on the two modules (Query Engine and Storage) running on GPU.

Storage module and Query Engine module are running on GPU, which play a key role in reducing memory access overhead and improving procedure efficiency. The Storage module uses arrays to describe information of rich metadata. In addition, these arrays are abstracted by metadata graph representation on GPU, which are arranged together in a contiguous memory chunk to reduce the time

of memory allocation, initialization, and management. In Query Engine module, the HPC rich metadata queries are turned into two basic array operations: the detecting operation and the gathering operation, which are performed on arrays stored in Storage module.

2.2 Metadata Graph Representation on GPU

As GRAM manages rich metadata by GPU-based property graph traversal and query, a suitable graph representation is needed. In GRAM, we design metadata graph representation with the GPU's benefits in mind. It is well known that, to take full advantage of GPU's high memory bandwidth, a coalesced memory access pattern is necessary, by which each cache line transmission contains more data required by the concurrent threads and then transferred to register files other than discarded. In other words, the data access instructions require less data traffic from memory to cache and register files. Considering the efficient and beneficial coalesced memory access pattern for GPU, we choose *Structure of Arrays* (SoA) instead of *Array of Structure* (AoS) as the layout of the property graphs, in which multiple arrays are used to hold the property values attached to vertices and edges of the graph. Comparing with AoS, SoA allows coalesced global memory accesses, which benefit GPU-based system. For the topological data of the graph, the commonly used *Compressed Sparse Row* (CSR) format is applied, simply because CSR format is easy to implement many graph algorithms (i.e., metadata graph traversal) in our vertex-centric programming model. Only CSR format can not meet our requirements for storing the properties of metadata, so more arrays are required in our graph representation.



(a) Metadata graph example (b) Corresponding arrays on GPU

Fig. 2. Metadata graph representation on GPU

Figure 2 shows a detailed example of the metadata graph representation. The topology of the graph is described by array *Edge_ptr* and array *Edge_dst* which construct the CSR structure. The property data including the IDs, names, types, values for vertices and edges are grouped into the other arrays that each array stores one simple data item for every vertice/edge. Importantly, these multiple arrays in Storage module are arranged together in a contiguous memory chunk to reduce the time of memory allocation, initialization, and management.

Algorithm 1 Detect(Frontier, PropertySet, Predicate and Marks)

```
1: Marks ← empty
2: for each Item in Frontier do
3:     if Marks.get(Item) == 0 then
4:         P ← PropertySet.read(Item)
5:         if Predicate(P) > 0 then
6:             Marks.set(Item)
7: return Marks
```

Algorithm 2 Gather(Frontier, Marks, NextFrontier, Collector)

```
1: NextFrontier ← empty
2: ResultMarks ← empty
3: for each Item in Frontier do
4:     if Marks.get(Item) > 0 then
5:         Coll ← Collector(Item)
6:         for each Result in Coll do
7:             if ResultMarks.get(Result) == 0  then
8:                 ResultMarks.set(Result)
9:                 NextFrontier.put(Result)
10: return NextFrontier
```

2.3 Metadata Graph Operations Model on GPU

As described above, the HPC rich metadata queries are translated into property graph traversals and finally mapped to array operations on GPU, which offers significantly better performance due to its high parallelism. In this section, we will introduce the array operations in GRAM's Query Engine module. Specifically, traversal and query in metadata graph are generalized into two basic array operations. By utilizing the array-based data layout, we focus on how the operations are performed on the arrays. Two types of basic operations are as follows:

- ***Detect whether properties conform to the filter criteria:*** During the detection of entities, one or more properties need to be filtered based on whether they conform to the criteria. Algorithm 1 presents the filter method. Different properties are filtered in a parallel or sequential manner. In addition, multiple filter criteria of properties can form a step of detection. Whether to parallelize the filtering depends on the procedure's efficiency. In addition, the multiple filters in each step is called combined filters. As shown in Figure 3, combined filters of 1-step detection consist of two filters, and combined filters of 2-step detection consist of three filters.
- ***Gather the vertices/edges with a qualified label:*** The gathering operation collects all vertices conformed to the filter criteria into a new frontier queue. Algorithm 2 shows how it works. The frontier queue may be a set of edges or a set of vertices. The whole process is iterative and convergent, the

frontier queue is either the intermediate results to process in next iteration or the correct results.

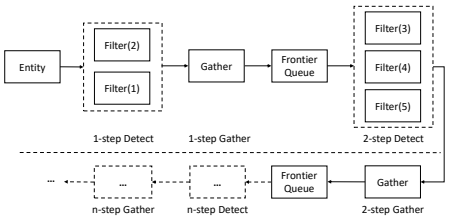
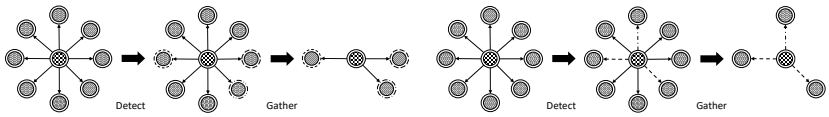


Fig. 3. The processing of n-step iterations



(a) The processing of vertex frontier array operations

(b) The processing of edge frontier array operations

Fig. 4. The processing of frontier array operations: detection and gathering on vertex/edge frontier

It is an iterative process to query metadata by operations on arrays. Furthermore, each step is composed of a detection operation and a gathering operation. Each iteration gets a new vertex/edge frontier queue. The detection and gathering operation of each iteration are shown in the Figure 4a and Figure 4b. There may be dependencies between steps of different iterations, but the data processed inside each iteration are independent. The synchronization step is achieved in BSP model. BSP operations leverage the parallelism of the GPU without any lock operations.

2.4 Metadata Operations Translating

GRAM manages rich metadata in a new fashion by introducing two basic operations on arrays. In Dai's previous research, GraphTrek [7] uses an asynchronous property graph traversal to query metadata. It defines property graph traversal operations based on an iterative query-building language. Several core methods in Query Interface module are applied to manage rich metadata. Query Interface module and Metadata Translating module cooperate to translate GraphTrek's main traversal methods into corresponding operations on arrays mentioned above. This section describes the translation in detail.

– *Vertex/Edge selector: v(), e()*. The vertex selector *v()* selects a specific set of vertices by setting a specific parameter, which represents the entry of property graph traversal to manage metadata in HPC systems. The edge selector *e()* selects a specific edge set from all the edges of a vertex by a specific label. Both of these two methods are very important and can select a specific subset based on the label of the vertex/edge. In the implementation of GRAM, these two selection methods are transformed to gather the vertices/edges with a qualified label on vertex/edge frontier.
– *Property filters: va(), ea()*. These two property filters have three parameters, property key, property values, and type of filter. Three types of filter include EQ, IN, and RANGE. Because each entity can have more than one property, multiple properties can be filtered by different property filters at the same time. In the implementation on the GPU, each filter turns to detect whether one of the properties conforms to the filter criteria.

Overall, metadata management is processed as graph traversal query, which in turn is translated into a series of operations on the corresponding arrays on GPU. The relationships between different entities and the storage schema for entities and properties are maintained on the CPU and then queried during the translation. After that, the main part of the query process which requires a large amount of data accesses and computation is dispatched to the GPU as a number of kernel functions launch that correspond to each operation on arrays. The CPU part manipulates a small amount of data structures that would involve dozens of to hundreds of successive random memory accesses, which could perform poorly on the CPU due to the low parallelism. The remaining part is suitable for GPU cause it can read and process the massive amount of data concurrently with its high bandwidth and computation power.

2.5 GRAM's Optimization

In this section, we describe two optimizations applied to the metadata management in property graph traversal and query fashion. The experimental results show that these two methods are effective. The design details are described as follows:

– *Parallel filtering*. In the detection phase, there are multiple types of filter criteria, which means that multiple properties of an entity are to be filtered. The filter criteria can be chosen to process serially or concurrently. Multiple filters are combined to detect concurrently. It is proved by our evaluations that the efficiency of concurrently detecting on combined filters is higher than that of serial selection.
– *Basic operations merging*. Metadata graph traversal and query is multi-step convergent iterations. The iteration in each step is based on two basic operations, detecting whether a vertex's property value conforms to some certain values, and gathering all the entities to a frontier queue as the next processing set. With the iteration of multiple steps merging, the performance of GRAM is improved.

Considering the requirements of HPC rich metadata management use cases, we design GRAM as graph traversal and query framework to manage rich metadata. The design and implementation of GRAM focus on managing rich metadata in a more efficient fashion.

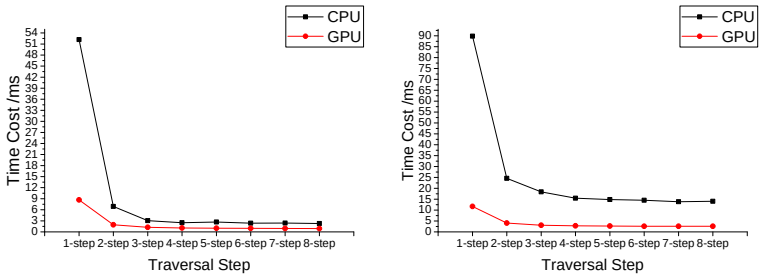
3 Evaluation

3.1 Experiment Environment and Datasets

Our experiments are conducted on a NVIDIA Telsa K20m GPU with 6GB main memory and 2688 CUDA cores. The GPU is installed on a machine with a 2.6 GHz Intel Xeon E5-2670 CPU and 64 GB memory.

To evaluate the metadata management capability of GRAM, we conduct experiments with GRAM on synthetic graphs. Our property graph datasets are generated as power-law metadata graphs by a RMAT graph generator [6], and during the generation, we also refer to Darshan log files [5]. In the graphs, vertices represent three kind of entities, user, job, and data files in Darshan log files, whereas edges reflect the relationships between them. We choose the same parameters as used in Dai's previous work [7] for the RMAT graph generator, that is $a = 0.45, b = 0.15, c = 0.15, d = 0.25$ for distribution parameters, 20 for graph scale and 16 for edge factor. The generated power-law graphs have moderated out-degree skewness, and each contains 2^{20} vertices and $16 * 2^{20}$ edges. Besides the graph topological data, we also generate several sets of uniformly distributed property values for both vertices and edges. When evaluating the 8-step graph traversal query, 8 sets of properties will be generated and used in each step for corresponding vertex and edge property checking.

3.2 Evaluating on Graph Traversal with Filters



(a) Metadata graph traversal with 1 filters

(b) Metadata graph traversal with 2 filters

Fig. 5. The each step time cost of 8-step metadata graph traversal with different filters

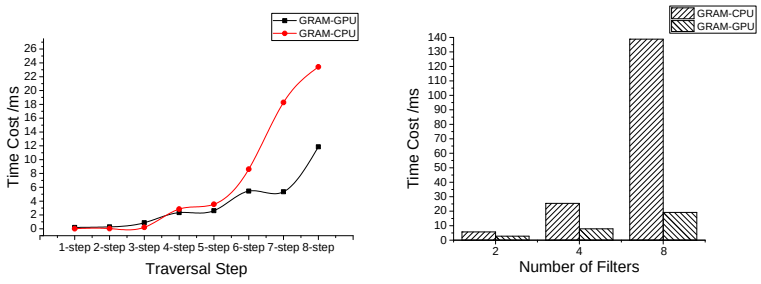
As described before, graph-based rich metadata management can be easily mapped to level-synchronous graph traversal operations, with extra filtering and path selection, which leads to more memory accesses and filtering computations than the traversal of normal graphs. Compared to traditional level-synchronous graph processing, using metadata graph to manage rich metadata requires to improve the performance of rich metadata queries through the high parallelism and high memory bandwidth of GPU. We compare the metadata queries on CPU and the metadata queries on GPU without considering the user scenarios in HPC systems. The metadata graph traversal begins with filtering all entities in this part of experiment. The number of filters determines the number of entities to process in next step. Furthermore, the number of entities is decreasing with the number of filters increasing. As shown in Figure 5, we change the number of filters in each step in metadata graph traversal. Figure 5a shows each step time cost of 8-step metadata graph traversal with 1 filters. The evaluations on CPU are conducted by 16 threads. As shown in Figure 5a, the time cost of metadata queries on CPU is 4.44 times lager than the time cost of metadata queries on GPU on average. In addition, with the number of filters increasing to 2, as shown in Figure 5b, the ratio is 6.43 on average. The efficiency of metadata graph traversal and query on GPU is better with more filters.

While serving metadata queries by property graph traversal, some properties will be filtered. As the evaluation results show, CPU has limitations to manage rich metadata in use cases in HPC fields. GPU's high memory bandwidth helps significantly in reducing the memory access latency and improving the efficiency of memory access. In addition, GPU's parallelism and storage resources can further improve the performance of metadata management.

3.3 Metadata Management Performance

We evaluate the performance of GRAM on the synthetic graph datasets. As we described above, HPC rich metadata management requests are translated into metadata graph traversal and query, the features of which are determined by rich metadata management use cases. Unlike level-synchronous traversal described in Section 3.2, rich metadata management use cases begin with a certain vertex, and the number of entities in frontier queue is increasing with the depth of the traversal hierarchy. The level of metadata traversal is not deep depending on the HPC metadata management scenarios, and not less than 3 steps in most cases. Actually, Dai's previous work [7] has found that rich metadata traversal are no more than 8-step graph traversal typically. Therefore, we perform 1 to 8 step metadata graph traversal to audit user in both GRAM-CPU and GRAM-GPU. The filtering probability and the scale of graph dataset in each step, which greatly affect the performance of the metadata management, are determined by the variation of user audit. The performance of GRAM's metadata traversal is shown in Figure 6a. The x-axis of Figure 6a illustrates the traversal steps, while the y-axis denotes to the total traversal time. If the traversal level is low, the number of entities to process is relatively small, and the time cost is too small to omit. Overall, we can see that GRAM based on GPU can significantly improve

the performance of traversal performance compared to graph traversal based on CPU.



(a) The performance of traversal (b) The performance of multiple detections

Fig. 6. The performance of rich metadata management

As described in Section 2.3, it is an iterative process to manage metadata by operations on arrays. Furthermore, one or more properties need to be filtered according to the criteria. We use multiple detections to realize filtering parallelism. The number of filters influence the traversal performance of GRAM, so we execute the performance of graph traversal by changing the number of filters. As shown in Figure 6b, we set the number of filters 2, 4, and 8 respectively, and the corresponding time cost grows with the increment of filter number. The GPU's advantage becomes more obvious when the filter number increases. We do not consider the filter number more than 8 due to usage scenarios.

4 Related Work

Using property graph to manage rich metadata is firstly proposed in Dai's previous work [9], and they have done many researches [7,8] in asynchronous property graph traversal for rich metadata management in HPC systems. Our work also translates rich metadata into one property graph, and uses property graph traversal and query to manage metadata. Dai's previous work [7,8] have focused on an asynchronous property traversal to manage metadata. In fact, the amount of data processed by metadata property graph traversal is large and property graph traversal requires more memory access. Our GPU-based property graph traversal framework for HPC rich metadata management can deal with the problems better.

Diverse property graph databases have been developed to manage property graph in recent years, such as Neo4j [23], DEX [2], G-Store [17], and Titan [4]. These property graph databases have been proposed to conduct property graph traversal and query, but the performance of these property graph databases in rich metadata management is limited. For example, Titan stores property

graphs based NoSQL storage systems like HBase [15] or Cassandra [18], in which all vertices are mapped to different rows; edges and properties are mapped to separate columns in the rows of the related vertices. In fact, because of HPC system's requirements of rich metadata management, traversal and query for property graphs need to be more efficiently supported.

There are many distributed graph processing frameworks. The search structure of Pregel [19], PowerGraph [12], GraphX [13] and other distributed graph processing frameworks is level-synchronous, just like breadth first search structure. These distributed graph processing frameworks have focused on different problems, while we focus on rich metadata management in HPC systems.

Harish and Narayanan [14] have given the first CUDA implementations of various graph algorithms. Merrill [20] et al. have implemented a scalable high-performance BFS graph traversal using CUDA. Their concern has been the optimization strategies of the GPU graph traversal, while our focus is the optimization strategies of property graph traversal and query on the GPU. Totem [11] is a CPU-GPU hybrid graph processing engine that overcomes the GPU memory limitations by assigning workloads on CPU cores and GPU cores. MapGraph [10] is a parallel programming framework on GPU, using dynamic scheduling and two-stage decomposition strategy to balance workload thread-divergence problems. CuSha [16] is a user-defined vertex-centric graph processing frameworks that can process large-scale graphs on a GPU. The concern of these graph processing frameworks are not property graph, while HPC metadata property graph processing is more challenging.

5 Conclusions

In this work, we manage rich metadata in property graph traversal and query fashion. Proper graph representation for the metadata graph on GPU is needed. Furthermore, there is lack of optimization techniques specifically for graph traversal to utilize the potentials of GPU. We propose GRAM, a GPU-based property graph traversal framework for HPC rich metadata management. GRAM uses property graph representation on GPU, by which metadata management is transformed to operations on arrays. In addition, we use two optimizations, parallel filtering and basic operations merging, to accelerate graph traversal. The performance comparison of metadata management confirms that GRAM achieves better performance than metadata management on CPU. In addition, our GPU-based graph traversal and query method achieves better performance than the traditional level-synchronous approach.

6 Acknowledgement

The work is supported by the National Key R&D Program of China (No. 2017YFC0803700), NSFC (No. 61772218, 61433019, U1435217), and the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032).

References

1. Blueprints, https://github.com/tinkerpop/blueprints
2. DEX, http://www.sparsity-technologies.com/
3. Property Graph, http://www.w3.org/community/propertygraphs/
4. Titan, http://thinkaurelius.github.io/titan/
5. Darshan Data (2013), ftp://ftp.mcs.anl.gov/pub/darshan/data/
6. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: SDM (2004)
7. Dai, D., Carns, P., Ross, R.B., Jenkins, J., Blauer, K., Chen, Y.: GraphTrek: Asynchronous graph traversal for property graph-based metadata management. In: CLUSTER (2015)
8. Dai, D., Chen, Y., Carns, P., Jenkins, J., Zhang, W., Ross, R.: GraphMeta: A graph-based engine for managing large-scale HPC rich metadata. In: CLUSTER (2016)
9. Dai, D., Ross, R.B., Carns, P., Kimpe, D., Chen, Y.: Using property graphs for rich metadata management in HPC systems. In: PDSW (2014)
10. Fu, Z., Personick, M., Thompson, B.: MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In: GRADES (2014)
11. Gharaibeh, A., Beltrão Costa, L., Santos-Neto, E., Ripeanu, M.: A yoke of oxen and a thousand chickens for heavy lifting graph processing. In: CLUSTER (2012)
12. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed graph-parallel computation on natural graphs. In: OSDI (2012)
13. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph processing in a distributed dataflow framework. In: OSDI (2014)
14. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: HiPC (2007)
15. Khetrapal, A., Ganesh, V.: HBase and Hypertable for large scale distributed storage systems. Dept. of Computer Science, Purdue University pp. 22–28 (2006)
16. Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N.: CuSha: Vertex-centric graph processing on GPUs. In: HPDC (2014)
17. Kumar, P., Huang, H.H.: G-Store: High-performance graph store for trillion-edge processing. In: SC (2016)
18. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (2010)
19. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: SIGMOD (2010)
20. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: PPoPP (2012)
21. Muniswamy-Reddy, K.K., Holland, D.A., Braun, U., Seltzer, M.: Provenance-aware storage systems. In: USENIX ATC. pp. 43–56 (2006)
22. Tanenbaum, A.S., Bos, H.: Modern operating system. Prentice-Hall, 4 edn. (2014)
23. Webber, J.: A programmatic introduction to Neo4j. In: SPLASH (2012)
24. Zhang, Q., Feng, D., Wang, F., Wu, S.: Mlock: building delegable metadata service for the parallel file systems. Science China Information Sciences pp. 1–14 (2015)
25. Zhao, D., Shou, C., Maliky, T., Raicu, I.: Distributed data provenance for large-scale data-intensive computing. In: CLUSTER (2013)