POSTER: High-Throughput GPU Random Walk with Fine-tuned Concurrent Query Processing

Cheng Xu[1], Chao Li[1], Pengyu Wang[1], Xiaofeng Hou[2], Jing Wang[1], Shixuan Sun[3], Minyi Guo[1], Hanqing Wu[4], Dongbai Chen[4], Xiangwen Liu[4]
[1]Shanghai Jiao Tong University, [2]Hong Kong University of Science and Technology
[3]National University of Singapore, [4]Alibaba Inc

Abstract

Random walk serves as a powerful tool in dealing with large-scale graphs, reducing data size while preserving structural information. Unfortunately, existing system frameworks all focus on the execution of a single walker task in serial. We propose CoWalker, a high-throughput GPU random walk framework tailored for concurrent random walk tasks. It introduces a multi-level concurrent execution model to allow concurrent random walk tasks to efficiently share GPU resources with low overhead. Our system prototype confirms that the proposed system could outperform (up to 54%) the state-of-the-art in a wide spectral of scenarios.

Keywords: Random Walk, GPU, Co-location

1 Introduction

Random walk has recently become a very common cloud workload in today's data centers due to the widespread deployment of graph neural networks (GNNs). It is responsible for distilling the low-dimension feature vectors from the large original graphs, which will be in turn used for various tasks such as node/graph classification, link prediction and recommendation systems. Since random walk can reduce the target graph size without sacrificing performance, it has been widely adopted as part of GNN applications [7].

We have witnessed a growing interest in specialized frameworks for optimizing random walk workload on both CPU and GPU [2, 5, 6]. GPU random work systems possess great potential to outperform CPU-based systems. The reason is that operations sampling different vertices within one task are independent of each other, requiring no communication [1]. Recent works have simulated the deployment of concurrent graph applications on various architectures [3, 4]. In this regard, the massive parallelism of GPUs allows GPU-based frameworks to span thousands of concurrent walkers.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577482

However, leveraging GPUs to perform random walk is non-trivial due to its intensive, complex memory access pattern. Current GPU-based frameworks all apply the execution pattern of CPU based-systems directly to GPUs where a single random walk task is highly paralleled[5]. While this model perform well on CPUs, it can lead to severe GPU store and lower the overall throughput. Random walk is a kind of memory-intensive workload and suffers from the imbalance of memory-to-compute bandwidth. Especially for graphs exceeding the GPU memory capacity, the low PCIe bandwidth between the CPU and GPU causes most GPU cores to remain idle to wait for the data, resulting in severe GPU underutilization. One GPU might be capable of serving concurrent random walk tasks from multiple GNN workloads if a well-designed space sharing technique is applied.

In this paper we show that it requires fine-grained concurrency optimizations and significant engineering efforts to develop a framework to harness massive random walk tasks. We employ a concurrent model to reduce stalled GPU cores. Through extensive experiments, we show that our framework demonstrates great performance and scalability.

Our contribution can be summarized as:

- We introduce a multi-level concurrent execution model for random walk tasks.
- We implement and verify a high-throughput framework for concurrent random walk on GPU.

2 Background

We choose *alias method* as our neighbour selecting method. It builds two tables at its preprocess stage: a probability table P and an alias table A to draw samples. Alias method can be executed in either *offline* or *online* mode. The *offline* mode first constructs an alias table for the whole graph, so that the execution stage can be executed in $O(1)$ at the expense of $O(d)$ preprocessing for each vertex. On the other hand, the *online* mode only constructs partial alias table when needed.

3 Concurrent Random Walk System

We design a novel concurrent execution model which enables fine-grained GPU space sharing. It takes both the execution mode and graph property into account to alleviate the problem of computational stall. As presented in Figure 1, our framework combines both mode-level concurrency and

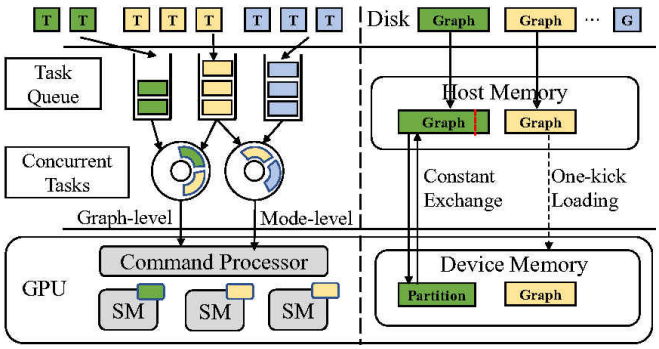

Figure 1. Multi-Level Concurrency.

graph-level concurrency for highest overall system throughput. This feature allows our system to adapt to a wide spectrum of scenarios.

Mode-level concurrency. *Offline* and *online* random walk tasks using alias method are complementary to each other. It can be seen by analyzing their time complexity and space complexity. In terms of time complexity, *offline* tasks work in a direct look-up table pattern, requiring only $O(1)$ for every vertex. Differently, *online* tasks need to first use $O(d)$ to construct that specific partition of alias table for the vertex on the fly, where d is the degree of that vertex. A small portion of CUDA cores can fulfill the computing resource requirement for *offline* tasks and the rest work of *online* tasks. In terms of space complexity, *offline* tasks rely on alias table construction, requiring $4 * |E| + 2 * |V|$ in space. For some large graphs, it means a difference of more than 15 GB in storage space. Concurrently running *offline* and *online* tasks can ease pressure on GPU memory and bandwidth, since *online* tasks do not require extra alias tables and only require $2 * |E| + |V|$ to store the graph data.

Graph-level concurrency. In graph-level concurrency, we smartly combine tasks of different graph sizes with the same mode. For *offline* tasks, there are abundant computing resources stalled for lack of memory bandwidth, especially for large graphs stored in the host memory. Our idea is to overlap data access to increase equivalent memory bandwidth and better utilize GPU cores. After finishing certain computing procedures, the GPU core needs to fetch graph data for the following process based on the result. The graph data to be visited may be currently stored in the GPU memory or the host memory, which is accessed differently. Data stored in host memory must first be transferred to GPU memory through the PCIe link, and this transfer process could be very lengthy. Therefore, we concurrently launch another kernel conducting random walk tasks on small graphs stored on GPU memory to exploit the on-board GPU memory resources while transferring these data, which significantly increases the equivalent memory bandwidth.

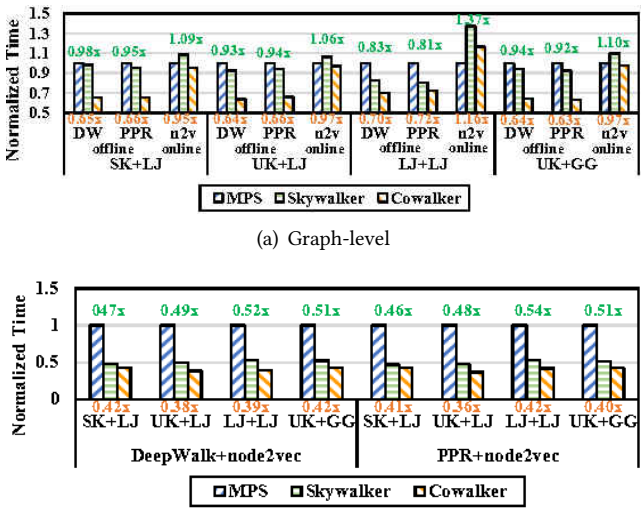

Figure 2. Normalized execution time of CoWalker with balanced query execution time compared with serial and MPS. *LJ*, *SK*, *UK* and *GG* are commonly used graph datasets.

4 Evaluation

Figure 2 shows the contribution of the two-level concurrency optimization used by CoWalker. Without CoWalker, the tasks are executed in serial. For graph-level concurrency, we execute `DeepWalk`; for mode-level concurrency, we execute `DeepWalk` and `PPR` with `node2vec`.

The MPS implementation fails to surpass the serial baseline in most scenarios, since the coarse-grained resource management strategy leads to interference between different tasks. It is especially obvious in Mode-level concurrency, where the MPS implementation is 1.87~2.11× slower. When it comes to the case of Graph-level concurrency, the execution time of MPS implementation in *offline* mode ranges from 1.02 ~1.08×. However, its execution time is less than the serial baseline in all cases in *online* mode. It even surpasses CoWalker in the case of *LJ+LJ* by 13%, showing superiority in walking on small graphs concurrently *online*.

The execution time of CoWalker is less than SkyWalker in all cases. With graph-level concurrency, the lowest execution time with CoWalker is only 67% of SkyWalker. For mode-level concurrency, CoWalker spends 75%~89% of the execution time compared to SkyWalker. It's because setting aside part of the computing resources will inevitably lower the performance of `node2vec` workloads.

5 Conclusion

We present CoWalker, a high-throughput GPU random walk framework enabling fine-tuned concurrent query processing. We hope that our study can spur further research in concurrent graph processing on heterogeneous systems.

References

[1] Peitian Pan and Chao Li. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *ICCD 2017*. 217–224. https://doi.org/10.1109/ICCD.2017.40

[2] Shixuan Sun et al. 2021. ThunderRW: An In-Memory Graph Random Walk Engine. *Proc. VLDB Endow.* 14, 11 (2021), 1992–2005. http://www.vldb.org/pvldb/vol14/p1992-sun.pdf

[3] Jing Wang et al. 2022. Excavating the Potential of Graph Workload on RDMA-based Far Memory Architecture. In *IPDPS 2022*. IEEE, 1029–1039. https://doi.org/10.1109/IPDPS53621.2022.00104

[4] Pengyu Wang et al. 2021. Grus: Toward Unified-Memory-Efficient High-Performance Graph Processing on GPU. *ACM Trans. Archit. Code Optim.* 18, 2 (2021).

[5] Pengyu Wang et al. 2021. Skywalker: Efficient Alias-Method-Based Graph Sampling and Random Walk on GPUs. In *PACT 2021*. IEEE, 304–317. https://doi.org/10.1109/PACT52795.2021.00029

[6] Ke Yang et al. 2019. KnightKing: a fast distributed graph random walk engine. In *SOSP 2019*. ACM, 524–537. https://doi.org/10.1145/3341301.3359634

[7] Rex Ying et al. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018).