BitGraph: A Framework For Scaling Temporal Graph Queries on GPUs

Alexandria Barghi
Department of Electrical and Computer Engineering
A. James Clark School of Engineering
University of Maryland
College Park, MD 20742
`abarghi@umd.edu`

Abstract

Graph query languages have become the standard among data scientists analyzing large, dynamic graphs, allowing them to structure their analysis as SQL-like queries. One of the challenges in supporting graph query languages is that, unlike SQL queries, graph queries nearly always involve aggregation of sparse data, making it challenging to scale graph queries without heavy reliance on expensive indices. This paper introduces the first major release of *BitGraph*, a graph query processing engine that uses GPU-acceleration to quickly process Gremlin graph queries with minimal memory overhead, along with its supporting stack, *Gremlin++*, which provides query language support in C++, and *Maelstrom*, a lightweight library for compute-agnostic, accelerated vector operations built on top of *Thrust*. This paper also analyzes the performance of BitGraph compared to existing CPU-only backends applied specifically to temporal graph queries, demonstrating BitGraph's superior scalability and speedup of up to 35x over naive CPU implementations.

1 Introduction

1.1 Background

When the field of Graph Theory was invented by Leonhard Euler, it was to solve a specific problem on a *structural graph*, which is defined as a graph consisting only of vertices and edges: $G = (V, E)$. Later, graph theorists and analysts began to use edge weight to quantify the strength of an edge between two vertices. But as mathematicians, data scientists, biologists, chemists, social scientists, and countless others began to apply graph theory to more complex problems, they realized that in many cases, vertices in the graph were defined not only by their relations to each other, but by specific variable properties unique to each vertex; for instance, age or height. And as the internet came of age, more and more applications also demanded edge properties beyond simple weight. Social networks linked entiries of all kinds to each other in a multitude of ways. Analyzing edges, predicting edge properties, creating edge-induced subgraphs, and similar operations became critical.

In the late 2000s, many of the people working on these new applications of graphs began to come together and formalize graph representations beyond classic structural graphs. The first graph standards, OWL [1] and RDF [2] had clear limitations and were atemporal [2]. Graph databases, which gained popularity during the 2010s, coalesced around an alternative model called the *Property Graph*. Property graphs allowed any graph element (vertex or edge) to have any number of properties [3]. In this model, representing time series data was now possible, as any node or edge could be tagged with a particular time. Mathematically, a property graph is $G_p = (V, E, F)$, where F in this expression is a set of functions $f(x) \in F$, defined for $x \in (V \cup E)$. [1] Each function can return anything in the universal set. In practice, most property graph processing engines limit the domain and range of these functions so that property graph data can be efficiently stored and retrieved.

The popularity of graph databases cannot be understated. There are established companies, such as *Neo4j* [4], startups such as *TigerGraph* [5] and *ArangoDB* [6], and even major players such as Amazon [7] and Microsoft [8] competing in the space.

Nearly all graph databases support at least one query language, an important tool in graph analytics that goes back to the days of OWL and RDF, when the first well-known graph query language, *SPARQL*, was conceived [9]. Graph query languages, such as Gremlin [10] and Cypher [11], are often the primary way of interacting with the graph, and the focus of this paper.

1.2 Graph Query Languages

Graph query languages are a means of asking questions about a property graph. These questions can range from the very simple (i.e. "How many friends does Bill have?") to those that are semantically and computationally complex (i.e. "What is the shortest path between Warehouse A and Warehouse B that does not go through Newark, NJ, and uses the minimum number of flights?"). Queries can easily be complicated by temporal features. For instance, certain nodes in the logistics network may not be available at given times, or routes between the nodes may actually correspond to individual flights or truck routes that depart and leave at specific times. Graph query languages can express these constraints very well.

Graph query languages can be very powerful in answering questions about temporal graphs. But as graphs grow larger, these queries become more difficult to process. GPUs have been used for years to accelerate graph algorithms, such as *Connected Components*, *Pagerank*, and *Breadth-First Search*, demonstrating that graphs are a strong target for the performance benefits of GPUs. Accelerating SQL or Spark queries on GPUs has also been mainstream for some time now, with clear benefits [12] [13]. But until now, no mainstream GPU-enabled graph query engine has ever been released.

1.3 History and Overview of the BitGraph Framework

The original, incomplete version of BitGraph was released by the author in 2019 [14], when GPU graph analytics was just beginning to gain steam, with projects such as Gunrock [15] [16], Hornet/-cuSTINGER [17] [18], and cuGraph [19] [20] all getting attention around the same time. BitGraph-2019 supplemented those projects as a graph query language engine that could offload certain processes to the GPU, boasting decent performance on small to midsize graphs. BitGraph-2019 was limited by the overhead of host-to-device (h2d) data transfer, since it inherited the CPU-based design of Java-Gremlin [14].

BitGraph-2023 (now a *framework* including the *BitGraph*, *Gremlin++*, and *Maelstrom* projects), is a complete reimagining of BitGraph-2019. It is still the only open-source GPU-supporting backend for Gremlin, and still includes Gremlin++, the only C++ frontend for Gremlin. However, BitGraph-2023 is now compute-agnostic, allowing users to mix and match CPU and GPU storage and execution. For vertex and edge properties, this mix-and-match capability is even more fine-grained, allowing allocation of specific properties to specific storage (CPU/GPU/UM [2]) and execution (device/host).

BitGraph-2023 is especially well-optimized for *ultra-sparse* property graphs, graphs where both the structure (vertices and edges) and properties (vertex properties and edge properties) are sparse. The graph structure is considered sparse when the number of edges (excluding multiple edges between the same two vertices) is much less than the maximum possible number of distinct edges: $|E| \ll |V|^2$ [22]. While there is no formal definition, in this paper, a graph property is considered sparse when it is present on fewer than half the total number of vertices.

The BitGraph framework was developed with other existing graph computing engines in mind, focusing primarily on accelerating graph queries, leaving other operations to frameworks such as PyTorch, Gunrock, cuGraph, or to graph databases.

[1] This definition is based on the author's experience with property graph processing systems and varies slightly from the definition in [3].

[2] UM refers to *unified memory*, which is an address space shared by the CPU and GPU [21].

2 Related Work

2.1 Gunrock

Gunrock [15] is one of the earliest and most well-known examples of a GPU graph analytics backend. It is designed for developing customized graph algorithms with zero accelerated computing or GPU programming knowledge. Gunrock exploits *frontier parallelism* [15]; each operation in Gunrock operates upon a frontier, just as each Gremlin step operates upon a set of traversers. The three main primitives (algorithmic building blocks) of Gunrock are *advance*, *filter*, and *segmented intersection* [15]. The four subtypes of advance primitives (Vertex-to-Vertex, Vertex-to-Edge, Edge-to-Vertex, and Edge-to-Edge) are also nearly identical to those in the BitGraph framework (Maelstrom/Gremlin++). The filter step in Gremlin++ is functionally identical to Gunrock's filter primitive, and Gunrock's segmented intersection primitive can be replicated by a combination of Gremlin steps in Gremlin++.

Gunrock, also like the BitGraph framework, supports optimizations similar to those done by Gremlin *traversal strategies*. Optimizations in Gunrock can include kernel fusion, which BitGraph supports more coarsely through traversal strategies.

Where Gunrock uses its own language to represent steps, BitGraph uses the more well-known Gremlin traversal language. BitGraph also relies on a huge suite of heavily-optimized Gremlin steps rather than the small set of primitives offered by Gunrock. Gunrock may ultimately support better performance for some algorithms, but its language does not have the expressiveness of Gremlin, and it doesn't support many basic query language features that data scientists and data analysts rely on.

2.2 cuSTINGER and Hornet

cuSTINGER is a data structure and framework for dynamic graph analysis [17]. Hornet is a successor to cuSTINGER offering superior performance, support for dynamic sparse matrices, and memory reclamation, with performance nearly identical or better to that of *CSR* [18]. BitGraph does not use Hornet, instead opting for a combination of the COO, CSR, and CSC matrix formats, which support BitGraph's *canonical representation*, and are better-suited for eventual multi-GPU support.

2.3 Blazing SQL and Dask-SQL

Blazing SQL was a SQL engine built on top of the NVIDIA RAPIDS ecosystem [12]. It was one of the first mainstream production frameworks for accelerating a query language (in this case, SQL) on GPUs, and helped show that that GPUs were both effective and cost-efficient for this purpose, delivering 20x speedup over Apache Spark for ETL queries. However, it has now been largely replaced by GPU-supported *dask-SQL* [13]. Like Blazing SQL, dask-SQL is built on top of the RAPIDS ecosystem, but takes advantage of the already-popular dask framework. It is also easier to scale dask-SQL beyond a single node, which is critical in large-scale applications. The BitGraph framework follows the same concept as these two projects, but accelerates Gremlin rather than SQL.

2.4 Lotan

Lotan links graph databases with graph neural networks (GNNs) [23]. GNNs are similar to convolutional neural networks (CNNs), but add graph structure. Training GNNs can be difficult due to the sheer data size of many production-scale graphs, which, without methods to generate subgraphs or samples of the full graph, are too large to train [23]. Lotan uses a query optimization system similar to the Gremlin traversal strategies to sample a graph stored in a graph database. But even with Lotan, the primary bottleneck for large-scale GNN training remains the query processing engines of graph databases [23]. While the BitGraph framework is designed for a general-purpose rather than a purely GNN-centric workload, it directly targets this key bottleneck.

3 Design of the BitGraph Framework

The BitGraph framework is comprised of *BitGraph*, *Gremlin++*, and *Maelstrom*, which were created and officially released for the first time in September 2023. Maelstrom is the bottom of the stack, providing a compute-agnostic (CPU/GPU) framework and API for vector computations, hash tables,

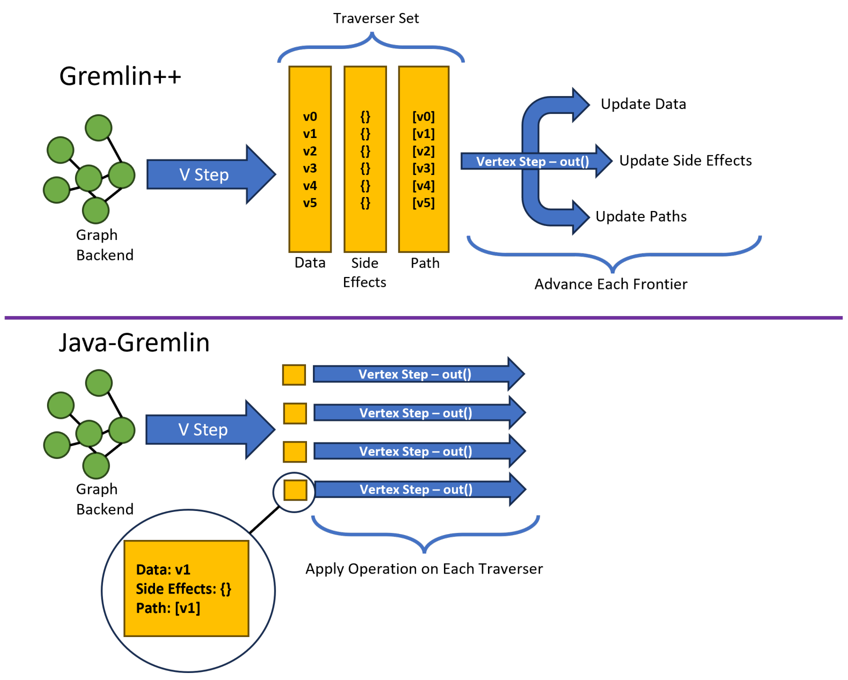

Figure 1: Gremlin++ vs. Java-Gremlin Traversal Comparison

and sparse matrices. Gremlin++ builds on Maelstrom to provide C++ language support for Gremlin, translating Gremlin steps into vector operations in the Maelstrom API. BitGraph defines the structure of the graph, including property storage, implementing the Gremlin++ API and using Maelstrom sparse matrices and hash tables to store graph data.

3.1 Staying Compute Agnostic

Graph Traversals in Gremlin are a series of operations upon *traverser* objects [24]. Traversers are atomic and can be shuffled around threads. However, shuffling these traverser objects runs into scaling issues, and even more so when writing a GPU implementation of Gremlin. To start, localizing data to a traverser object sacrifies spatial locality. Searching through or aggregating data across traversers will cause a high number of page faults. This penalty is compounded when using unified memory, which the BitGraph framework supports, since page faults may require transferring data through PCI-e. The Java-Gremlin traverser implementation is also heavily reliant on pointers and function calls, which increases memory fragmentation overhead, and would require many separate device memory allocation calls, which force synchronization with the host.

Supporting the traverser concept without sacrificing performance was critical. The solution was to reimagine Gremlin steps as vector operations, making traversers a virtual concept. Instead of localizing data to traverser objects, a structure called a *traverser set* manages three structures: *traversed objects*, *side effects*, and *path information*. These three structures globalize the data that traditional Gremlin localizes to each traverser (Figure 1). Gremlin++ still supports traverser semantics, but under the hood, is performing vector operations on traverser sets. This is the case regardless of where the traverser set is stored or where it is being processed, making it compute and storage agnostic. Enabling this was the primary objective of Maelstrom.

3.2 Overview of the Maelstrom API

Maelstrom provides a simple yet powerful API for vector operations. Inspired by libraries like *PyTorch* [25], Maelstrom uses the same API for all operations regardless of where data is being computed or stored. In nearly all cases, it can properly infer the optimal location (CPU/GPU) for computation, and transparently dispatch the appropriate code path. Maelstrom is also designed to eventually support multi-GPU, and potentially even multi-node processing with zero API change.

Because the rest of the BitGraph framework uses Maelstrom, all that is needed to enable future multi-GPU processing is the creation of that backend in Maelstrom.

Maelstrom also has APIs for two other key data structures used by the BitGraph framework, hash tables and sparse matrices. These APIs are designed to work with the Maelstrom vector API, and like the vector API, are designed to eventually be extended for multi-GPU processing. The hash table API is built on top of the cuCollections dynamic map API [26]. The sparse matrix API is written from scratch, and is designed specifically to support frontier parallelism as is used throughout the BitGraph framework. It also takes advantage of the Maelstrom vector API's type erasure capabilities to allow usage with a wide variety of types, unlike the cuSPARSE API [27] it largely replaces.

3.3 Query Processing in Gremlin++

Gremlin++ provides support for the Gremlin language in C++, using Maelstrom to handle its processing. The semantics of Gremlin++ are nearly identical to those of standard OLAP Gremlin, making migration easy for Gremlin-Java users. Gremlin++ is divided into two parts, the structure API, and the traversal API, mirroring Java-Gremlin. The structure API defines the basic components of a property graph: vertices, edges, and properties. The traversal API defines the Gremlin query language and how it is processed, by converting a Gremlin query into its internal representation (Gremlin Steps), running the just-in-time optimization process (traversal strategies), and then calling either functions in Maelstrom or in the structure API to execute the optimized steps. Functions that are part of the structure API have to be implemented by backends (i.e. BitGraph) since they require manipulating the internals of the graph, which Gremlin++ isn't aware of.

3.4 BitGraph Structure

BitGraph is the top of the BitGraph framework, implementing the Gremlin++ API and using Maelstrom to define its sparse matrix and property table structure. BitGraph is lightweight, comprised mostly of calls to the Maelstrom API. This makes it compute agnostic, compatible with Maelstrom's type erasure, and easy to extend. BitGraph has two components: the graph structure, which is stored as a Maelstrom sparse matrix, and the graph properties, which are stored as Maelstrom hash tables.

3.4.1 Sparse Matrices in BitGraph

Depending on the operation being done, BitGraph stores the graph in a canonical COO (coodinate list), CSR (compressed sparse row), or CSC (compressed sparse column) format. When modifying the graph, the canonical COO format is used. This format is a standard COO matrix sorted by insertion order. Using canonical COO format, adding or removing vertices or edges takes best case $O(1)$ time and worst case $O(|E|)$ time. Canonical COO format also saves memory by not requiring an additional $O(|E|)$ vector to store edge IDs.

Computing an adjacency query on the canonical COO matrix is of complexity $O(|T||E||V|)$, where *T* represents the set of vertices in the adjacency query. Using a CSR/CSC matrix to process the query, in contrast, has a far superior time complexity of $O(|T|)$, and memory usage can be reduced by doing the conversion in-place. In order to recreate the canonical COO matrix when it is needed, the edge permutation has to be stored, so canonical CSR/CSC format includes this permutation vector, for a total memory usage of $O(|E| + |V|)$. Given that canonical COO has memory $O(|E|)$, and that generally $|E| \gg |V|$, we can approximate the total memory usage as $O(|E|)$ for both canonical COO and canonical CSR/CSC. Therefore, converting to canonical CSR/CSC prior to adjacency queries has a clear performance benefit without sacrificing memory usage.

All conversion to and from matrix formats is done automatically. Since users interact through the Gremlin++ API, and write queries in Gremlin, they do not have to worry about the internals of the graph structure, including matrix format.

3.4.2 Vertex and Edge Properties

Vertex and edge properties are implemented through Maelstrom hash tables, which allow users to store properties in device, host, managed, or pinned memory. For properties that are frequently accessed and updated during queries, device storage, if feasible, is the preferred option. For most

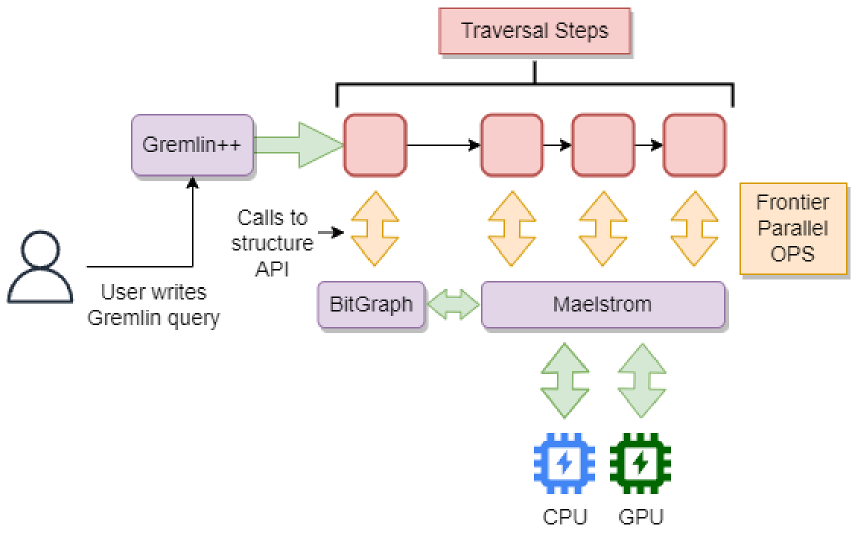

Figure 2: Life of a Gremlin Query in the BitGraph Framework

other properties, managed memory is preferred, since it will automatically move pages back and forth between device and host storage to save valuable device memory at a moderate performance cost.

3.5 Putting it All Together: End-to-End Life of a Query

When a user writes a query in Gremlin, it is converted into Gremlin steps. For instance, *g->V(39).out().next()* is a query that finds vertex 39, and returns the first of its outgoing edges. This is broken into two steps and one finalization operation. The first step is a *V Step*, an instruction to retrieve vertex data from the graph and put it into the traverser set (frontier). The second step is a *Vertex Step*, and is an instruction to retrieve outgoing edges from a vertex. In this traversal, the finalization operation is *next*, which gets the first element of the traverser set and returns it.

Once a query has been transformed into steps, the resulting steps are optimized using *traversal strategies*, which are similar to compiler optimizations. For instance, one strategy combines the limit step with steps that support running with a limit, which can reduce redundant computations.

Finally, the query is processed. Gremlin++ interprets the final set of steps, and when necessary, will call API functions that dynamically dispatch to BitGraph. Each step updates the traverser set (frontier), passing it on to the steps that follow.

Figure 2 shows how a Gremlin query is optimized, transformed into vector operations, and computed.

4 Performance

Through GPU acceleration, low-level optimization, and its traverser set structure, BitGraph achieves impressive speedup over Java-Gremlin. BitGraph was benchmarked against the standard Gremlin in-memory graph backend, TinkerGraph [28]. Unless otherwise indicated, benchmarks were run on an overclocked Intel i7-12700K CPU and NVIDIA RTX A6000 GPU.

4.1 Connected Components

The connected components algorithm is good benchmark for understanding how users who don't know how to write CUDA kernels can achieve superior performance over what they could do with an existing Gremlin frontend and backend. For this benchmark, a variation of the algorithm described in [29] was used. The Gremlin traversal for this algorithm is in Figure 3.

The connected components algorithm is useful for temporal graph processing, as it allows for partitioning a larger graph into smaller components that can then be classified based on their temporal behavior. This is a common approach used for *graph classification*.

```
g->V().property("cc", id()).iterate();
g->V().property("old_cc", values("cc")).iterate();
size_t diff = 1;
while(diff > 0) {
    diff = g->V()
        .property("old_cc", values("cc"))
        .property("cc",
                      union({
                               both().values("old_cc"),
                               values("old_cc")
                      }).min()
                )
        .elementMap({"cc", "old_cc"}).where("cc", neq("old_cc"))
        .count().next();
}
```

Figure 3: Gremlin Traversal for Connected Components

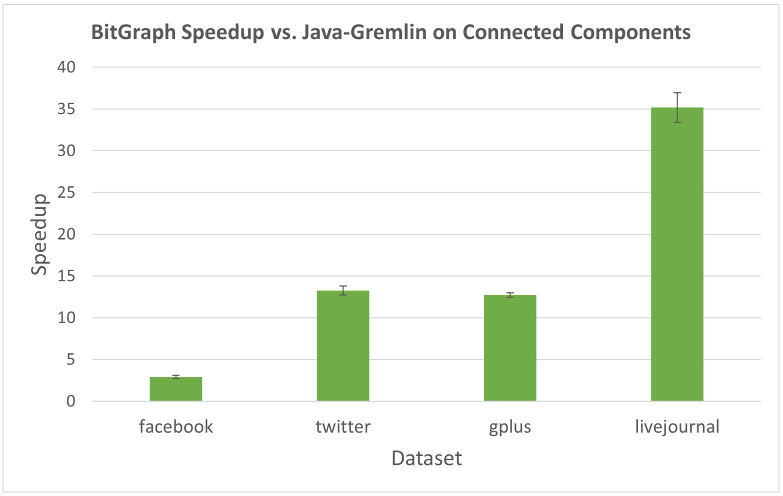

Figure 4: Connected Components Performance Benchmarks

This benchmark used the following datasets: *facebook-combined* (4K vertices, 90K edges) [30], *twitter-combined* (80K vertices, 2.4M edges) [30], *gplus-combined* (100K vertices, 30M edges) [30], and *livejournal* (5M vertices, 70M edges) [31] [32]. These were all obtained from SNAP [33]. Results are shown in Figure 4. [3] BitGraph performed strongly on these benchmarks, due to its CSR structure and GPU minimum aggregation, delievering an impressive 35x speedup on the largest dataset.

4.2 Temporal Shortest Paths

Shortest paths algorithms are among the most commonly-used graph algorithms. The Gremlin language offers a simple means of constructing temporal shortest paths queries.

Consider the following problem: *Suppose I have a package that I need to transfer through my logistics network. Whenever a package arrives at a destination facility, it needs to first be sorted overnight, before it can be sent on a plane the next day. How many nodes in the logistics network can this package reach within four days, provided I know the current location of the package?*

This problem can be translated into a Gremlin query, as shown in Figure 5.

For this benchmark, the tgbl-flights dataset (60K vertices, 70M edges) [34] was used. It was obtained from the Temporal Graph Benchmark (TGB) project [35]. The tgbl-flights-1M dataset is a subset of tgbl-flights consisting of the first 1 million edges. Results are shown in Figure 6. [4] BitGraph achieved a speedup of 2.7x on flights-1M and 1.8x on the full dataset, once again benefiting from the CSR

[3]The livejournal benchmark was run on an AMD EPYC 7452 CPU and NVIDIA RTX A6000 GPU due to the high RAM usage of Java-Gremlin on this dataset.

[4]The flights-full benchmark was run on an AMD EPYC 7452 CPU and NVIDIA RTX A6000 GPU due to the high RAM usage of Java-Gremlin on this dataset.

```
g->V(v_start)
  .repeat(
      property("visited", 1)
      .elementMap({"last_time"})
      .outE().has("time", gremlinxx::P::lte(time_end))
      .elementMap({"time"})
      .where("time", gremlinxx::P::gt("last_time"))
      .as("last_e")
      .values("time").as("last_time")
      .select("last_e").inV().hasNot("visited")
      .elementMap({"name"})
      .as("v")
      .select("last_time").min(ScopeContext(Scope::local, "name"))
      .select("v")
      .property("last_time", select("last_time"))
  ).iterate();
auto v_total = g->V().has("visited").count().next();
```

Figure 5: Gremlin Temporal Shortest Paths Traversal

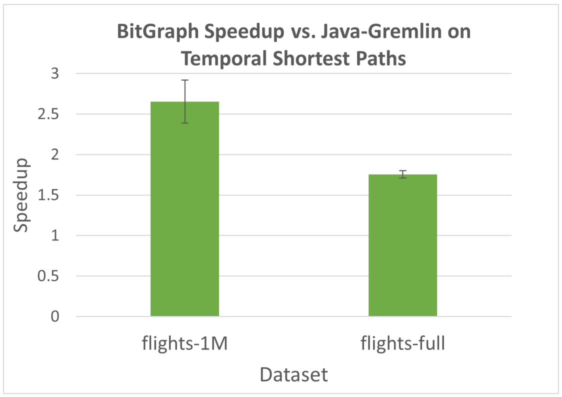

Figure 6: Temporal Shortest Paths Performance Benchmarks

structure and minimum aggregation on the GPU, although the Repeat Step did cause some additional d2h transfers that limited the potential speedup. Improving the Repeat Step in Gremlin++ to avoid d2h copies would likely bring the speedup closer to the 35x achieved in the previous benchmark.

5 Applications and Future Work

5.1 More Gremlin Features

Gremlin++ currently provides the most important Gremlin steps, but there are still many more steps that Gremlin++ does not yet support, including some that would simplify or accelerate temporal queries. A good example is the *SimplePath* Step [24], which automatically filters out cycles. Gremlin++ also currently does not support Python bindings, which is a targeted feature for a future release. Support for Python and other languages could eventually be provided by a Gremlin server [24], which is how Java-Gremlin is able to interface with other languages.

5.2 Just-in-Time Optimization and Dispatching

Gremlin steps can be combined with each other through traversal strategies, reducing the number of kernel launches, and eliminating redundant operations. Other optimizations could use CUDA streams to run concurrent queries or loop iterations. BitGraph could also support dispatching to Gunrock, or even faster implementations of specific algorithms such as *Connected Components* or *Pagerank*.

5.3 Multi GPU

Extending Maelstrom to a multi-GPU environment would allow Gremlin++ to process queries with many billions or even trillions of traversers, potentially across multiple disjoint graphs.

Acknowledgments and Disclosure of Funding

The author did not receive any third-party compensation for this work. The author is currently employed by the NVIDIA corporation as a full-time software engineer. All computing resources used for development and testing were either owned by the author or rented from a neutral third-party cloud provider.

The author is currently a doctoral student at the University of Maryland advised by Professor Manoj Franklin, Dept. of Electrical and Computer Engineering, A. James Clark School of Engineering

The author would also like to acknowledge the following people:

- *Emily Q. Cox*, for proofreading and graphics support

References

[1] W3C. (2012) Owl 2 web ontology language document overview (second edition). [Online]. Available: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/

[2] ——. (2014) Rdf 1.1 concepts and abstract syntax w3c recommendation 25 february 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Comput. Surv.*, vol. 50, no. 5, sep 2017. [Online]. Available: https://doi.org/10.1145/3104031

[4] Neo4j Inc. (2023) Neo4j graph database & analytics. [Online]. Available: https://neo4j.com/

[5] TigerGraph. (2023) Tigergraph: Drive competitive advantage with the enterprise-scale graph data platform for advanced analytics and machine learning. [Online]. Available: https://tigergraph.com

[6] ArangoDB. (2023) Arangodb, the database for graph and beyond. [Online]. Available: https://arangodb.com

[7] Amazon Web Services. (2023) Amazon neptune. [Online]. Available: https://aws.amazon.com/neptune/

[8] S. Andrews, M. Brown, J. Codella, E. Mutta, M. Bouameur, and M. Ho. (2023) Welcome to azure cosmosdb. [Online]. Available: https://learn.microsoft.com/en-us/azure/cosmos-db/introduction

[9] W3C. (2013) Sparql 1.1 query language w3c recommendation 21 march 2013. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[10] M. Rodriguez, "The gremlin graph traversal machine and language," in *Proceedings of the 15th Symposium on Database Programming Languages*, ser. Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2815072.2815073

[11] 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*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1433–1445. [Online]. Available: https://doi.org/10.1145/3183713.3190657

[12] A. Ocsa, "Sql for gpu data frames in rapids accelerating end-to-end data science workflows using gpus," in *LatinX in AI Research at ICML 2019*, 2019.

[13] R. Gelhausen. (2022) Announcing the general availability of dask-sql on gpus. [Online]. Available: https://medium.com/rapids-ai/announcing-the-general-availability-of-dask-sql-on-gpus-5abe5312c5d5

[14] A. Barghi, "Gremlin++ & bitgraph: Implementing the gremlin traversal language and a gpu-accelerated graph computing framework in c++," Master's thesis, University of Maryland, ProQuest, Ann Arbor, MI, 2019.

[15] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017. [Online]. Available: http://escholarship.org/uc/item/9gj6r1dj

[16] Regents of the University of California. (2021) Gunrock: Gpu graph analytics. [Online]. Available: https://gunrock.github.io/docs

[17] O. Green and D. Bader, "custinger: Supporting dynamic graph algorithms for gpus," 09 2016.

[18] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," 09 2018.

[19] T. Hricik, D. Bader, and O. Green, "Using rapids ai to accelerate graph data science workflows," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–4.

[20] NVIDIA Corporation. (2023) Welcome to rapids graph documentation. [Online]. Available: https://docs.rapids.ai/api/cugraph/stable/

[21] M. Harris. (2017) Unified memory for cuda beginners. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[22] P. E. Black, *Dictionary of Algorithms and Data Structures*. NIST, 2019.

[23] Y. Zhang and A. Kumar, "Lotan: Bridging the gap between gnns and scalable graph analytics engines," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 2728–2741, 2023.

[24] Apache TinkerPop. (2023) Tinkerpop documentation. [Online]. Available: https://tinkerpop.apache.org/docs/3.7.0/reference/

[25] PyTorch Foundation. (2023) Pytorch. [Online]. Available: https://pytorch.org/

[26] Y. Wang, J. Hemstad, D. Jünger, R. Maynard, P. Taylor, and S. Kang. (2023) cuCollections. [Online]. Available: https://github.com/NVIDIA/cuCollections

[27] NVIDIA Corporation. (2023) cusparse. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[28] Apache TinkerPop. (2023) Tinkergraph-gremlin. [Online]. Available: https://tinkerpop.apache.org/docs/3.7.0/reference/#tinkergraph-gremlin

[29] J. Soman, K. Kishore, and P. J. Narayanan, "A fast gpu algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[30] J. Leskovec and J. Mcauley, "Learning to discover social circles in ego networks," *Advances in neural information processing systems*, vol. 25, 2012.

[31] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 44–54.

[32] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[33] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[34] M. Strohmeier, X. Olive, J. Lübbe, M. Schäfer, and V. Lenders, "Crowdsourced air traffic data from the opensky network 2019–2020," *Earth System Science Data*, vol. 13, no. 2, pp. 357–366, 2021. [Online]. Available: https://essd.copernicus.org/articles/13/357/2021/

[35] S. Huang, F. Poursafaei, J. Danovitch, M. Fey, W. Hu, E. Rossi, J. Leskovec, M. Bronstein, G. Rabusseau, and R. Rabbany, "Temporal graph benchmark for machine learning on temporal graphs," 2023.