ABSTRACT

Title of Thesis:	GREMLIN++ & BITGRAPH: IMPLEMENTING THE GREMLIN TRAVERSAL LANGUAGE AND A GPU-ACCELERATED GRAPH COMPUTING FRAMEWORK IN C++
	Alexander B. Barghi, Masters of Science, 2019
Thesis Directed By:	Professor Manoj Franklin

This thesis consists of two major components, Gremlin++ and BitGraph. Gremlin++ is a C++ implementation of the Gremlin graph traversal language designed for interfacing with C++ graph processing backends. BitGraph is a graph backend written in C++ designed to outperform Java-based competitors, such as JanusGraph and Neo4j . It also offers GPU acceleration through OpenCL .

Designing the two components of this thesis was a major undertaking that involved implementing the semantics of Gremlin in C++, and then writing the computing framework to execute Gremlin's traversal steps in

BitGraph, along with runtime optimizations and backend-specific steps. There were many important and novel design decisions made along the way, including some which yielded both advantages and disadvantages over Java-Gremlin. BitGraph was also compared to several major backends, including TinkerGraph, JanusGraph, and Neo4j. In this comparison, BitGraph offered the fastest overall runtime, primarily due to data ingest speedup.

GREMLIN++ & BITGRAPH: IMPLEMENTING THE GREMLIN TRAVERSAL LANGUAGE AND A GPU-ACCELERATED GRAPH COMPUTING FRAMEWORK IN C++

by

Alexander Barghi

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Masters of Science
2019

Advisory Committee:
Professor Manoj Franklin, Chair
Yavuz Oruc
Bruce Jacob

Dedication

I dedicate this thesis to my father, for his unconditional support and encouragement, and for inspiring me to become an engineer. I could not be here without his guidance.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction & Background

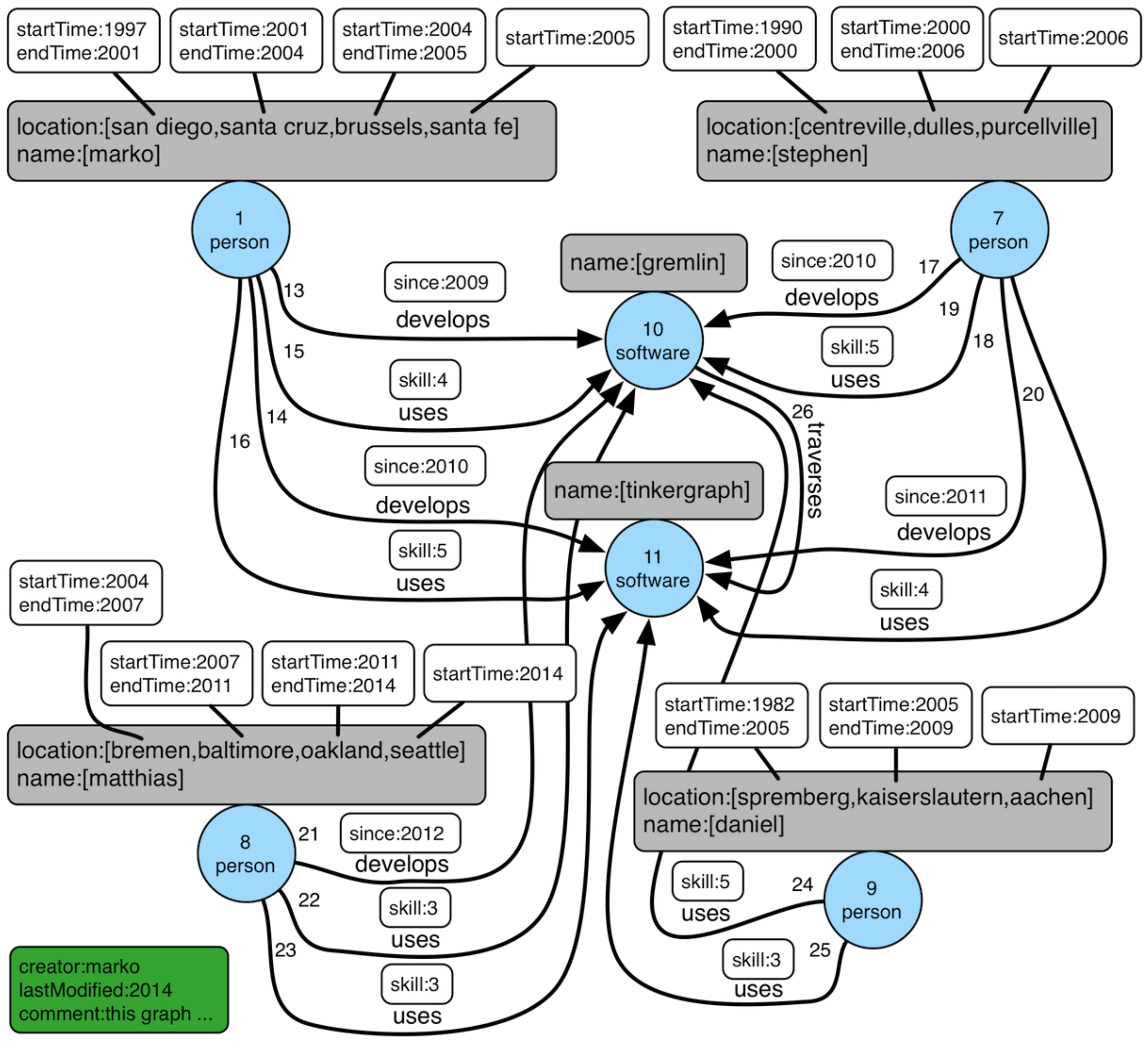
Gremlin

Origin of Gremlin

The Gremlin language [1] came out of the Apache TinkerPop project. Initially, it was part of the *Pipes* subproject, but TinkerPop 3 merged Pipes with several other subprojects to create Gremlin, the primary product of Apache TinkerPop [2]. In this thesis, *Gremlin* usually refers to the Gremlin language rather than the combined project, which includes other features related to but separate from the language itself.

The Gremlin Language

Gremlin is a *domain-specific language* for interacting with property graphs [1]. A *property graph* (Figure 1) is an extension of traditional digraph $G = (V, E)$ that includes a special function $\lambda : \left((V \cup E) \times \Sigma^*\right) \rightarrow (U \setminus (V \cup E))$ [1]. λ maps each element in the graph onto an object in the universal set (excluding graph elements), forming the final property graph $G = (V, E, \lambda)$ [1]. Property graphs are used to express various networks in a compact structure friendly to traditional databases. The fundamental unit of computation in Gremlin is the *traverser*, a structure that can hold any object in the universal set [1]. Each traverser also has a *bulk*, a number representing its multiplicity, and a *sack*, a collection of temporary variables stored in the traverser [1]. Mainstream property graph databases (*backends*) supporting Gremlin include Neo4j[1] [3], Amazon Neptune [4], and JanusGraph [5]. TinkerGraph is the reference implementation of a Gremlin-supporting backend [6], and is maintained by the authors and maintainers of the Gremlin language.

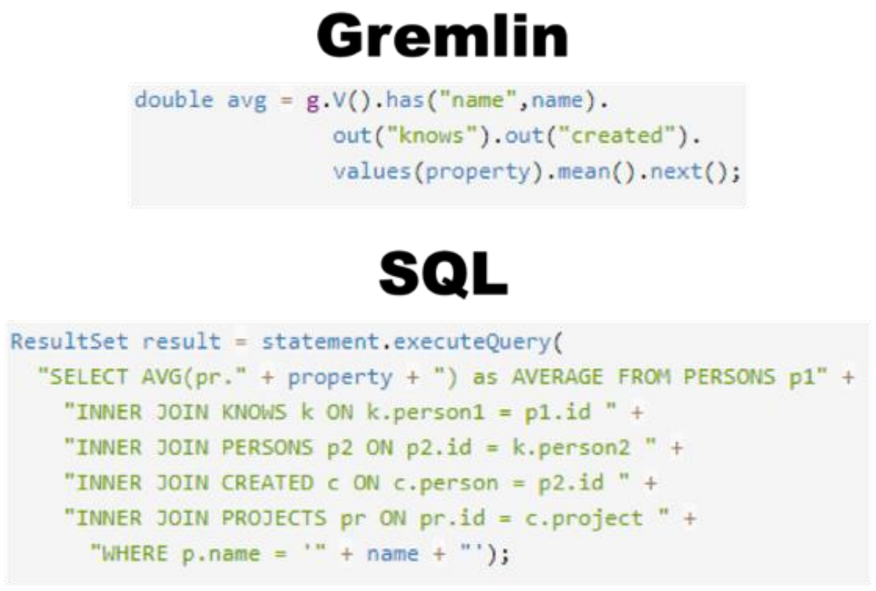

startTime:1997 endTime:2001
startTime:2001 endTime:2004
startTime:2004 endTime:2005
startTime:2005
startTime:1990 endTime:2000
startTime:2000 endTime:2006
startTime:2006
location:[san diego,santa cruz,brussels,santa fe] name:[marko]
location:[centreville,dulles,purcellville] name:[stephen]
1 person
7 person
name:[gremlin]
10 software
since:2009 develops
since:2010 develops
skill:4 uses
skill:5 uses
traverses
name:[tinkergraph]
11 software
since:2010 develops
skill:5 uses
since:2011 develops
skill:4 uses
13 14 15 16 17 18 19 20 21 22 23 24 25 26
startTime:2004 endTime:2007
startTime:2007 endTime:2011
startTime:2011 endTime:2014
startTime:2014
startTime:1982 endTime:2005
startTime:2005 endTime:2009
startTime:2009
location:[bremen,baltimore,oakland,seattle] name:[matthias]
location:[spremberg,kaiserslautern,aachen] name:[daniel]
8 person
9 person
since:2012 develops
skill:3 uses
skill:3 uses
skill:5 uses
skill:3 uses
creator:marko
lastModified:2014
comment:this graph ...

[1] Neo4j is a registered trademark of Neo4j Inc.

Figure 1: The Crew, an example of a Property Graph [7].

Why use Gremlin?

There are two key reasons for adopting Gremlin. Firstly, Gremlin is designed to support many host languages while still remaining *first-order*. A *first-order* DSL is one that can use the constructs of the host language. Figure 2 shows a comparison to SQL, which is not first-order. Secondly, Gremlin is general enough to support many backends, from the simplest in-memory backends to massive distributed databases. This comes from the use of *just-in-time compilation* (known as *traversal strategies*) [1] to optimize Gremlin code at runtime and convert it to backend operations.



Gremlin

```
double avg = g.V().has("name",name).
                out("knows").out("created").
                values(property).mean().next();
```

SQL

```
ResultSet result = statement.executeQuery(
  "SELECT AVG(pr." + property + ") as AVERAGE FROM PERSONS p1" +
    "INNER JOIN KNOWS k ON k.person1 = p1.id " +
    "INNER JOIN PERSONS p2 ON p2.id = k.person2 " +
    "INNER JOIN CREATED c ON c.person = p2.id " +
    "INNER JOIN PROJECTS pr ON pr.id = c.project " +
      "WHERE p.name = '" + name + "'");
```

Figure 2: Gremlin vs. SQL

Contributions of the Thesis

Gremlin++

Overview

Gremlin++ is the first low-level implementation of a Gremlin interpreter. I created it in 2018 and continue to develop and maintain it. Gremlin++ is designed to be as compatible with Gremlin++ as possible while adhering to C++14 standards and requirements. This includes allowing the user to use any property value in the universal set, and supporting traversal strategies (just-in-time compilation). This year, I released Gremlin++ as an open-source product (Apache 2.0 License) and made it available on GitHub[2].

Motivation

There are substantial advantages to writing a Gremlin interpreter in C++, including the speed benefits of a low-level language and support for C and C++ backends. Prior to Gremlin++, the only Gremlin interpreter was Java-Gremlin, which could be interacted with through many languages via the Gremlin server, but which fundamentally operated in the JVM. [7] Gremlin++ is a huge step forward in this regard. Additional motivations for choosing C++ included its ubiquity on HPC systems and mature support for GPUs through CUDA [8] and OpenCL [9].

[2] Gremlin++ on GitHub: https://github.com/bgamer50/gremlin-

BitGraph

Overview

Alongside Gremlin++, I also developed BitGraph, the first-ever low-level language backend for Gremlin. Its purpose is to store and manage a graph structure, and interface with Gremlin through Gremlin++ (Figure 3). BitGraph interfaces with Gremlin++ through its own interpreter, which runs on top of the Gremlin++ interpreter. It also offers support for GPU acceleration through OpenCL [9], which it is also a pioneer in. I have made BitGraph open-source (Apache 2.0 License) and available on GitHub[3].

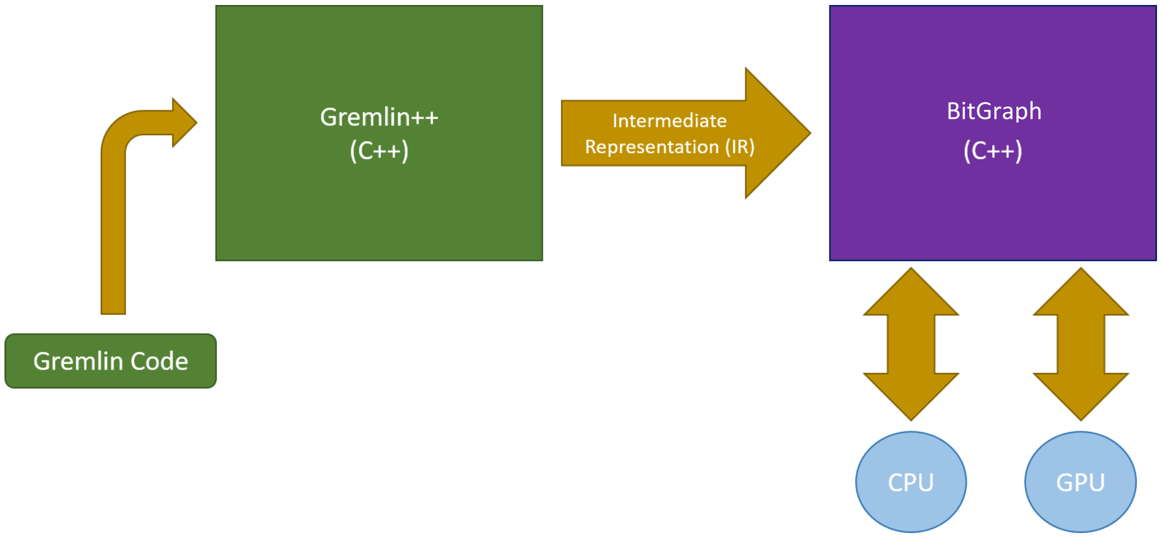


Figure 3: Relationship between Gremlin, Gremlin++, and BitGraph

Motivation

The motivations for BitGraph are similar to those for Gremlin++. Firstly, serving as a backend for Gremlin++ demonstrates its utility. Secondly, because BitGraph is written in a low-level language, it has the potential

---

[3] BitGraph on GitHub: https://github.com/bgamer50/bitgraph

to outperform existing JVM backends. Additionally, BitGraph's support for GPUs opens the door to innovations in parallelism and optimization of Gremlin code.

Chapter 2: Design & Development of Gremlin++

Initial Design

Language

As discussed previously, I chose C++ as the language for Gremlin++. While C offered many of the same features and benefits as C++, it did not offer function chaining, a key requirement to implement Gremlin. Thus C++ was the clear choice. The C++14 standard was selected with the needs of BitGraph in mind. There are some key syntactical differences between C++ and Java; in Gremlin++ the most evident of these differences is the use of *pointers*.

Boost Library

The Boost Library was used primarily for the <boost/any.hpp> header, which includes the boost::any container and boost::any_cast function [10]. These were used to handle dynamic types in order to support properties of any type in Gremlin++. It is left up to backends whether to offer support for specific types.

Structure API

The structure API is based off the TinkerPop 3 structure API [11], which is technically not part of Gremlin. However, some basic structure is needed to support steps that specifically deal with graph elements such as Vertices, Edges, and Properties. The Gremlin language is designed to avoid use of the structure API by the user, but within the interpreter, it allows more steps to have a "default" implementation, meaning that less work has to be done by the backend. Today, Gremlin++ translates raw Gremlin to an intermediate representation, relying on backends like BitGraph to provide an interpreter. However, my eventual goal is to include a fully-functional default interpreter in Gremlin++ to simplify backends and allow backend implementers to focus on specific optimizations. This goal would not be possible without a structure API.

The structure API in Gremlin++ covers two graph elements (*Vertex, Edge*) and one higher-level abstraction (*Element*). Vertex and Edge inherit from Element, mirroring the Java API. At this time, Gremlin++ only supports properties on Vertices (*VertexProperty*). *VertexProperty* inherits from the generic *Property* interface, which any future edge properties will also implement. Having an interface for properties is crucial since it allows them to be stored predictably in Traversers by the Gremlin's various *property* steps.

Implementation of Gremlin Semantics

Handling of Dynamic Types

Dynamic types, variables whose type is unknown at compile time, are a significant challenge to implement in a statically-typed language like C++. However, they are key to supporting Gremlin's support for virtually any property value.

An illustration of how Gremlin uses dynamic typing is in how the various *Property* and *Id* steps allow a diverse range of backend behavior. TinkerGraph, for example, supports nearly any valid Java type, including user-defined types [6]. TinkerGraph also allows several types of Ids, including Strings, Longs, Ints, or UUIDs [6]. Java-Gremlin uses Java generics to support this feature [12].

Dynamic types are also key to Gremlin's traversers. Java-Gremlin supports traversers that can contain *any* object, and does not construct new traversers when avoidable [12]. Once again, this feature is supported by Java generics [12].

Templates in C++14 do not support dynamic types. Unlike Java generics, which use type erasure, templates actually generate a separate class for each type. This makes it difficult, if not impossible to use them to implement properties, since there is no such thing in C++ as a variable of a class whose template specification is unknown. The typical answer to this problem has been to use a void* pointer, but this results in nearly unreadable code due to the number of casts required and lack of type information. It also gives up type safety, which is an important feature of statically-typed languages. Therefore, use of void* is a non-ideal solution.

Thankfully, the Boost Library [13] offers a powerful solution for handling dynamic types. The *Any* container in Boost is a container that can hold primitives, objects, and pointers [10]. It offers type safety through the *boost::any_cast*() method, and also supports empty containers (an extremely useful feature when indexing properties). boost::any also supports safe argument passing and safe copying [10].

My initial version of Gremlin++ relied on void* pointers, which resulted in a long development time for individual traversal steps, since these had to be managed carefully. After switching to boost::any to store property values and objects held by traversers, development time for new traversal steps was cut dramatically. The code was also much easier to understand, especially for end-users, who would no longer have to worry about potential undefined behavior when casting void* pointers.

Function Chaining

Function chaining (Figure 4) is a technique in object-oriented programming languages where the output of a function is an object with immediately-callable functions [14]. As stated previously, function chaining is required to write a Gremlin interpreter. One roadblock to using function chaining in C++ is that C++ references come with more restrictions regarding their creation. Users must be able to dynamically allocate Graph Traversals in order to support anonymous traversals. Different types of backends must also be allowed to have their own traversals, whose type is unknown at compile time to Gremlin++ (as it is backend-agonstic). As a result, one key change from Java-Gremlin is the use of pointers with traversals instead of references. For instance, *g.V().out().has("name", "joe").next()* in Java-Gremlin becomes *g->V()->out()->has("name", "joe")->next()* in Gremlin++, where *g* is a graph traversal source.

Without Function Chaining
b = a.foo();
c = b.bar();
d = c.foo();

With Function Chaining
b = a.foo().bar().foo();

Figure 4: Function Chaining Illustration

Anonymous Traversals

Anonymous traversals, sometimes called recursive traversals, are a key feature of Gremlin. They are necessary when implementing the language to handle steps whose arguments might be valid Gremlin [1]. Take, for instance, the Gremlin traversal *g.V().property("d", out().count()).iterate()*, which finds and saves the out-degree of each vertex in the graph. In Gremlin-Java, this becomes *g.V().property("d", __.out().count()).iterate()*. The __ in that traversal is a generator that produces anonymous Graph Traversals, which are themselves interpreted and optimized.

Gremlin++ deals with anonymous traversals by providing its own version of __. Instead of using a specialized class, Gremlin++ uses a macro that produces a Graph Traversal using the default constructor. The resulting traversals are not attached to a graph, and can be passed as arguments to steps, leaving the attachment to be handled at runtime by the backend. Due to the current implementation of traversal strategies, attachment to the graph and optimization of the traversal can only be done in an ad-hoc manner by the backend.

Implementation of the Interpreter

Traversal Strategies

Traversal strategies are translations that "rewrite a traversal (with typically, though not necessarily, the same semantics as the original traversal)" [1]. Gremlin relies on traversal strategies to handle backend-specific code and just-in-time optimizations [1]. The Java implementation of Gremlin allows backends to register their backend-related strategies with the compiler. In Gremlin++, the process is currently more ad-hoc. Rather than specific strategies, a Graph Traversal executes the *getInitialTraversal()* method, which applies all strategies known to the traversal. Through polymorphism, different backends (with their own Graph Traversals) can extend the behavior of the default *getInitialTraversal()* method.

The Default Graph Traversal

Gremlin++'s "default" Graph Traversal is the core of the library. It provides all functionality of the Gremlin language. Each step has one or more corresponding methods that can be chained together to form valid Gremlin. This is analogous to the default Graph Traversal in Java-Gremlin. The various methods often make use of boost::any, especially for steps that deal with properties. Graph Traversals in Gremlin++ also hold a list of steps which are ultimately executed after *getInitialTraversal()* is called by the backend.

Graph Traversals also have finalization steps, such as the *iterate()*, *forEachRemaining()* and *next()* methods. These steps bring about the execution of the traversal. Unlike Gremlin-Java, Gremlin++ leaves it to the backend to handle the execution process. This will likely be fixed as the library matures.

Implementation of Key Steps

Has Step

The Has Step supports a variety of predicates, including *equals*, *less than or equal to*, *greater than*, etc [15]. Each of these predicates operates on a Vertex or Edge and returns true if the Traverser should continue, and false if it should not. In Gremlin-Java, the *P* class acts as a generator for these predicates. It automatically produces predicates for any object, usually by calling methods from the Object or Comparable interfaces in Java [16]. In C++, these interfaces don't exist. Furthermore, the use of boost::any hold any property type complicates predicate testing since there needs to be a function that handles the comparison ready at compile time. To get around this issue, Gremlin++ uses its own *P* class, but returns lambda expressions rather than non-executable bytecode as in Java. This makes it easy for users to use their own comparison functions with the Has Step. The downside of this approach is that it makes Has Steps harder to inspect at runtime, complicating the optimization process. In the future, Gremlin++ might switch to a Java-style non-executable predicate for the Has Step and better support the Filter Step for dealing with user-created classes.

Add Property Step

The Add Property step modifies a Graph by adding a property to a Vertex or Edge [17]. If the property already exists, it is modified or replaced depending on its cardinality (single, list, or set) [17]. Only Vertices support cardinalities other than *single* [18]. Add Property Steps in Gremlin++ work directly on Elements, taking in an Element pointer and modifying the Element accordingly. It is possible to create a backend-specific Add Property Step which extracts the key, value, and cardinality of the Add Property Step and replaces it with a new backend-specific step.

Gremlin++ also supports property addition using anonymous Graph Traversals. Support of this feature is left to the backend interpreter since it requires dynamic creation of backend-specific traversal objects.

Property Step

Property Steps access either a Property container or the actual value of a Property [19]. Much like the Add Property Step, the Property Step directly accesses an Element. Property Steps contain a type (value or property) and list of property keys. Currently, Gremlin++ does not provide a default execution of the Property Step. Instead, it leaves its execution to the backend interpreter. This will change when standalone traversal strategies are properly implemented.

Graph Step

Graph steps access specific vertices or edges in the graph [20]. In Gremlin++, this is left up to the backend since Gremlin++ has no way of knowing how to access Vertices on a backend without the Structure API. In the future, a default implementation using the Structure API will likely be provided along with the option of replacing it using traversal strategies. The default Graph Step in Gremlin++ holds the type (vertex or edge) and requested elements, if any.

Add Edge Step

The Add Edge Step in Gremlin++ supports all features of the Java-Gremlin implementation of this step, including modulation with *from()* and *to()* [21]. Like the Graph Step and Property Step, it is largely a container in Gremlin++, with the backend interpreter doing most of the work.

Chapter 3: Design & Development of BitGraph

Graph Structure

Overview

Gremlin++ only provides interfaces for the various graph elements and properties, leaving it to the backend to handle their storage and management. Determining how the graph structure was stored was the first part of designing BitGraph. BitGraph uses a vertex-centric approach to graph storage, meaning that most data is stored in the vertices, including edge information. It also uses sequential identifiers as opposed to randomly-generated UUIDs for differentiating elements in the graph.

However, although most data lives on the vertices, BitGraph supports indexing on properties, a key feature of nearly all property graph backends. Indexing allows fast querying and filtering by property values through a lookup table instead of a full vertex scan, reducing property filter and query time from $O(N)$ to $O(1)$. All indexes are stored at the graph level.

Vertices

BitVertexes, the BitGraph Vertex implementation, are the core of the graph in BitGraph. Each BitVertex has a unique 64-bit ID, along with a property store and edge list. The edge list is broken into two lists, one for incoming edges and the other for outgoing edges. Both lists are implemented with C++'s std::list.

Edges

BitEdges are the BitGraph Edge implementation. They contain two pointers (one to the out-vertex and one to the in-vertex). They also have a unique 64-bit id. BitEdges do not currently support properties or labels, but this is planned in the future.

Properties

Each BitVertex stores a map of properties, keyed by the property key, using the C++ std::map container. The values of these properties are contained in VertexProperty objects provided by the Gremlin++ API. BitGraph does not support multiproperties (property key with multiple values) or metaproperties (properties of properties). In order to support any property type, the VertexProperty objects are specialized with boost::any. This means the user will need to cast the properties when directly accessing them. It is assumed the user knows what type their properties are; if not, boost::any provides methods for determining type [13].

Indexes

Indexes are an important feature of most property graph backends. Property graph support diverse graphs with elements that may represent very different objects. Take for instance the “Modern” graph [22] that ships with TinkerGraph [23] (Figure 5)[4]. This graph connects people with the software they created and the people they know. It has two types of vertices and two types of edges. Suppose a user wants to know the languages created by each user. They would use the traversal in Table 1, which outputs a map of each person to the software they created (Table 2).

```
g.V().hasLabel('person')
        .project('person', 'software created')
        .by(values('name'))
        .by(out('created').values('name').fold())
```

Table 1: Person to Software Traversal.

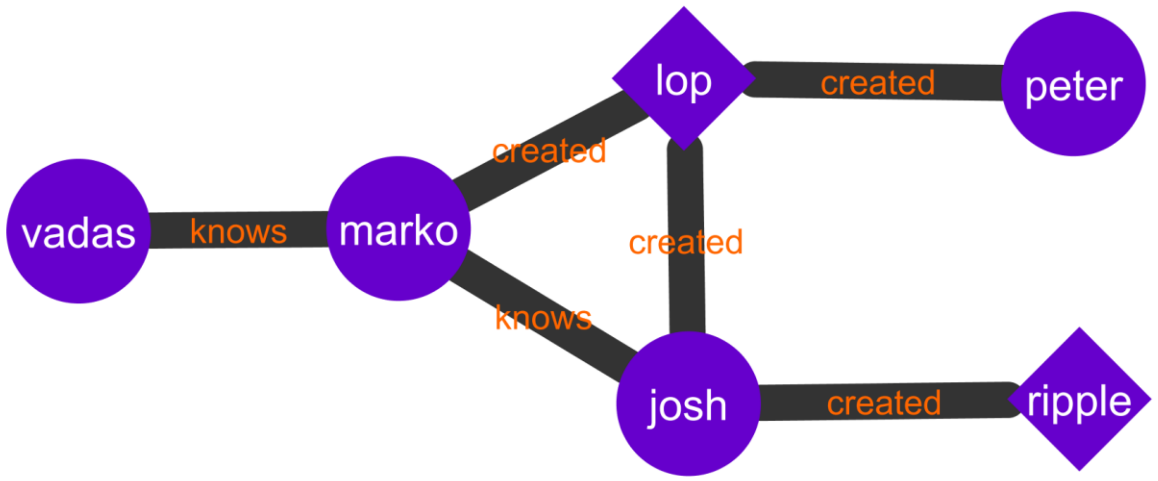


Figure 5: The Modern Graph.

[4] Created with Cytoscape. [28]

```
==>[person:marko,software created:[lop]]
==>[person:vadas,software created:[]]
==>[person:josh,software created:[ripple,lop]]
==>[person:peter,software created:[lop]]
```
Table 2: Traversal output on Modern Graph.

The above example shows the utility of property and label querying in Gremlin. For a small graph, it is trivial to check the label and property of each edge, but on the CPU, it is an $O(N)$ operation. Indexes, along with a backend-specific index step, can reduce this to an $O(1)$ operation through a lookup table. Instead of looking at each element to be filtered, a lookup table of each property value to the list of elements with that property allows this to be done in a single step. The only exception to this rule is when path information or side effects need to be preserved after a filter. In this scenario, each element needs to be matched with a Traverser, which is still an $O(N)$ operation, but is a good target for parallelism.

As mentioned previously, indexing in BitGraph is implemented using a lookup table and backend-specific step. The lookup table is a custom hash table[5] that can index any property using the boost::any container so long as the user provides a hash function. For most common types of properties (numbers, strings), this is very simple as C++ has default implementations for these functions. For custom classes, the user can

[5] The table is implemented using a dynamic array (std::vector) and resolves collisions through linear probing.

write their own hash function and pass it to BitGraph upon index creation. Each key value is also associated with any elements with that value. Indexes are element-specific; they can lookup either vertices or edges, not both. This is done to allow future support for multiproperties on vertices [18], an optional feature of Gremlin not currently supported by BitGraph. There is also an id index that allows lookup of Vertices by unique id; this is implemented using the default C++ unordered map rather than a BitGraph index.

Backend-Specific Interpreter

Backend-Specific Steps

Backend-specific steps and just-in-time optimizations are tightly coupled in Gremlin, just as they are in Gremlin++. BitGraph uses several such steps, most notably the Index Step, which performs Has Steps using the index if it exists for the property being filtered on. Backend-specific steps are also used to handle GPU operations if the user is using a GPU Traversal. Currently, the three supported GPU operations are Filter, Min, and Has.

Just-In-Time Optimizations

BitGraph's just-in-time optimizations convert certain steps to Index Steps and the traversal *g.E()* to the vertex-centric traversal *g.V().outE()*, in addition to the optimizations provided by Gremlin++. The Index Step in

BitGraph directs the interpreter to search for graph elements using indexes. This is a constant-time operation. When the optimizer detects a Graph Step followed by a Has Step, and knows there is an index on the property examined by the Has Step, it replaces the two steps for a single Index Step. The replacement of the Edge Graph Step with a Vertex Graph Step and Vertex Step is done since BitGraph is vertex-centric and uses Vertices to manage edges.

Traversal Execution

In Java-Gremlin, nearly all the interpretation is done in the default traversal [11]. Since Gremlin++ does not currently have a means of using standalone traversal strategies with the default interpreter, BitGraph does most interpretation. This is likely to change as Gremlin++ matures. When the user calls one of the finalization steps (iterate, next, etc.), interpretation begins. Each Graph Traversal begins with some sort of start step, which accesses the starting elements of the Traversal. The interpreter begins by getting the elements requested by the start step and putting them into Traversers. Those Traversers are then passed to the next step. Calls to the various functions associated with step execution are aggressively inlined using the *inline* keyword and the O3 compile option in gcc.

Some Traversal Steps may contain their own traversals, such as the Add Property Step, which allows the property to be added to come from a

Traversal. In this case, BitGraph's Traversals provide special methods for converting an anonymous default traversal into an executable backend Traversal at runtime. The Traversal that is created matches the type of the creator Traversal (either a CPU or GPU Graph Traversal). Just-in-time optimizations are applied to these traversals as well, although in some cases not all optimizations can be applied (i.e. start step optimizations). Special methods also exist to properly copy Traversers to hand off to the newly created Traversals, and properly delete Traversers once the new Traversals have finished.

GPU Traversals

Relationship to CPU Traversals

In BitGraph, there are two types of Graph Traversals, CPU Graph Traversals and GPU Graph Traversals. Because not all operations can be done on the GPU, GPU Graph Traversals inherit from CPU Graph Traversals. GPU Graph Traversals rely on the CPU interpreter, but override some of the execution methods. There are also GPU-specific steps which replace the default steps during the just-in-time optimization process[6]. GPU Graph Traversals must be created from a GPU Traversal Source. To get a GPU Traversal Source, the user first gets a CPU Traversal Source using *graph.traversal()*, then calls *withGPU()* to

[6] GPU Graph Traversals have their own getInitialTraversal() method that first calls the CPU's optimization method, then does GPU optimizations.

convert it to a GPU Traversal Source. GPU Traversal Sources handle the setup and bookkeeping operations needed for GPU access, and should be reused whenever possible to avoid the overhead associated with creating them. Figure 6 illustrates the relationship between the graph and various traversal sources and traversals in BitGraph.

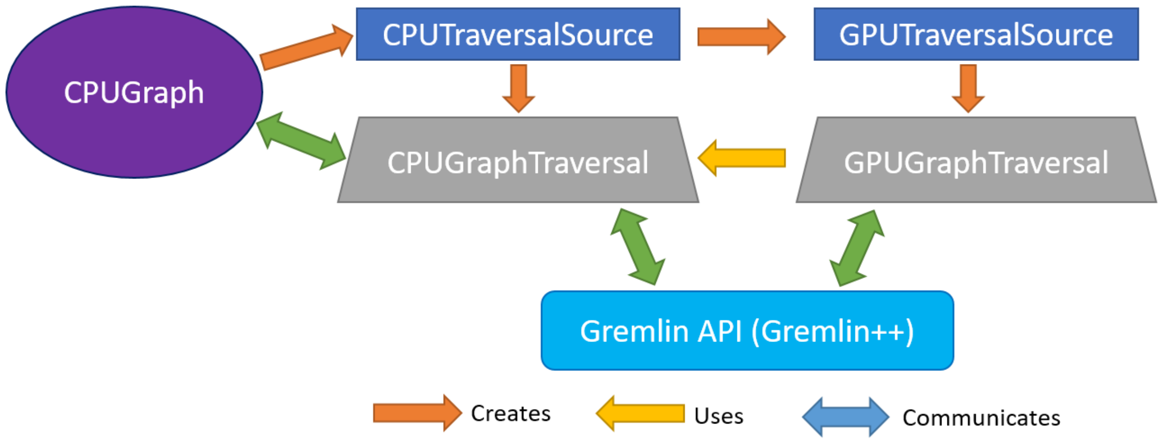


Figure 6: BitGraph Organization

Overhead Reduction

The overhead of accessing a GPU can be a hassle; as a result, BitGraph tries to save as much of a GPU context as possible in the traversal source so that repeated uses of the traversal source do not require additional kernel compilation. Each GPU Graph Traversal Source contains compiled kernel code and references to the current GPU context and device. These are all obtained upon creation of the traversal source. Because BitGraph is not transactional, there are few if any use cases for creating multiple traversal sources.

GPU-Specific Steps

The GPU currently supports the Filter and Min steps through the OpenCL library [1]. The Filter Step is similar to the Has Step, but supports any predicate. The Min Step finds the minimum of the contained objects in a Traversal and passes through its value in a new Traverser. To execute Has Steps on the GPU, the Q class exists as an analogue of the P class on the CPU. This deviation from Java-Gremlin exists due to the earlier design decision to promote the arguments to the Has Step to full predicates. The Q class generates predicates that run on the GPU, such as *Q::eq*, which produces a predicate that compares equality.

The two steps that currently exist as special GPU steps, the Filter Step and Min Step, have pre-written GPU kernels that get compiled when the traversal source of the current traversal is created. Table 3 shows the code for the Filter Step. Table 4 shows the code for the Min Step. Figure 7 shows how traversers on the CPU are sent to the GPU for execution.

```
__kernel void filter(__global ulong* expected, __global ulong* values, __global bool* result) {
    const int i = get_global_id(0);
    result[i] = values[i] == expected[0];
}
```

Table 3: Filter Step Kernel Code

```
__kernel void minimum(__global ulong* sz, __global ulong* values) {
    ulong size = sz[0];
    const int i = get_global_id(0);
    int j;
    for(j = 1; j < size; j=j*2) {
        if(i \% (2*j) == 0) {
            int m = values[i];
            if(!(i + j >= size || m < values[i+j])) m = values[i+j];
            values[i] = m;
        }
        barrier(CLK_GLOBAL_MEM_FENCE);
    }
}
```
Table 4: Min Step Kernel Code

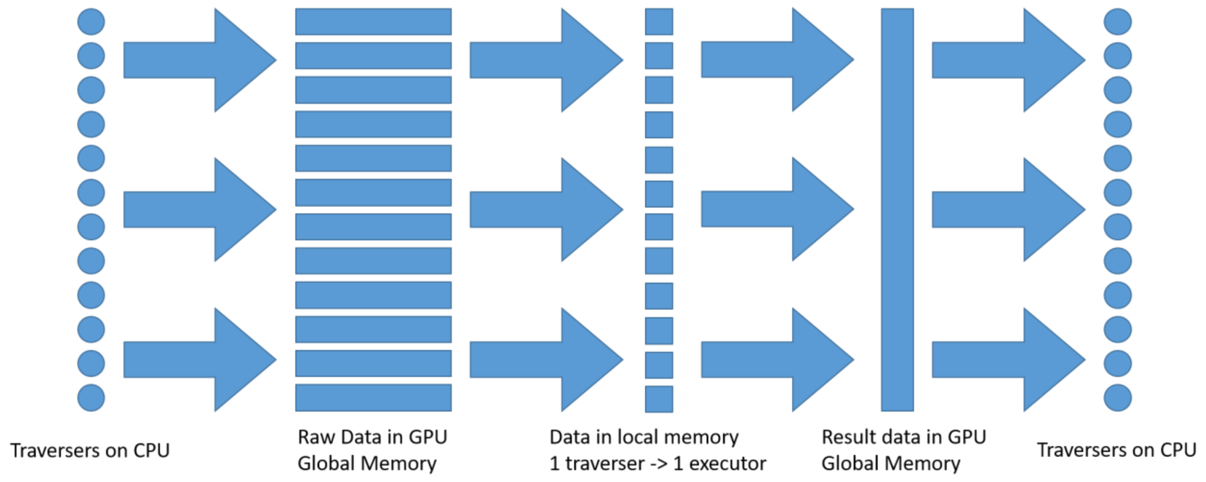

Traversers on CPU
Raw Data in GPU Global Memory
Data in local memory 1 traverser -> 1 executor
Result data in GPU Global Memory
Traversers on CPU

Figure 7: GPU Step Execution

Chapter 4: Benchmarks & Evaluation Environment

Overview

Algorithms

The connected components algorithm is a key algorithm in the field of graph analytics. The goal of the algorithm is to identify each set of vertices in a graph such that each contained vertex can reach any other in the set [24]. The Apache TinkerPop project recognized this and recently added an OLAP implementation of connected components directly callable using Gremlin [25]; however, not all backends support OLAP. The version of connected components used to test and benchmark BitGraph and the other backends is written in Gremlin and does not use OLAP. It initializes the component id to the current Vertex id, then repeatedly finds the minimum of the current component id and neighbors' components ids until there is no change. Because each iteration is identical, the benchmarks in this paper use a single iteration, referred to as "cc1x". The Gremlin code for this traversal is shown in Table 5.

```
g.V().property('cc', id()).iterate()
g.V().property('cc', coalesce(both(), identity()).values('cc').min()).iterate()
```

Table 5: The cc1x Traversal

The degree centrality algorithm is another common graph algorithm used to identify important nodes in a network. It is an $O(|V|)$ algorithm that counts the number of edges on each vertex. There are three variations of this algorithm, in-degree, out-degree, and total degree [26]. In-degree measures the number of incoming edges on each vertex [26]. Out-degree measures the number of outgoing edge on each vertex [26]. Total degree measures the number of edges in either direction on each vertex [26]. The out-degree algorithm was benchmarked in this paper. The Gremlin code for its traversal is shown in Table 6.

```
g.V().property("d", out().count()).iterate()
```

Table 6: Gremlin Traversal for Degree Centrality (out-degree)

Datasets

Two datasets were chosen for comparison across backends. Both came from the Stanford Large Network Dataset Collection, a product of the Stanford Network Analysis Project (SNAP) [27]. The first was the ego-Facebook dataset, which contains 4,000 vertices and 90,000 edges. The second was the ego-Twitter dataset, which contains 80,000 vertices and 2,000,000 edges.

Backends

For the purposes of benchmarking, BitGraph using CPU traversals and BitGraph using GPU traversals were treated as separate backends. Three other major backends were selected to compare against BitGraph. The first was TinkerGraph, which is the reference implementation of a

Gremlin backend [23]. TinkerGraph is in-memory, primarily relying on Java's HashMap class. The second was JanusGraph, a Gremlin backend designed for large data which supports several database backends and an in-memory backend [5]. Only the in-memory backend was benchmarked in this paper. The third was Neo4j [3], one of the most well-known graph backends in the industry, which has support for Gremlin through the Neo4j-Gremlin library.

System

The system used for the benchmarks was powered by an AMD Ryzen 7 2700x 3.7 GHz 8-core processor, 32 Gigabytes of DDR4 RAM, and an AMD Radeon RX 580 GPU with 8 GB of internal memory. All datasets tested fit into both the CPU RAM and GPU internal memory. JVM startup time was not included in total runtime. The C++ programs were compiled to native code and did not have any startup time.

Chapter 5: Results & Analysis

Data Ingest

Results

The ego-Twitter and ego-Facebook datasets are simple edge lists. The ids in the dataset were treated as properties, and ingested into the graph under the "name" property. Each backend used a series of Gremlin traversals to ingest the data. Figure 8 and Figure 9 show the results on each backend for ego-Facebook and ego-Twitter, respectively. Appendix A contains the entire table of results.

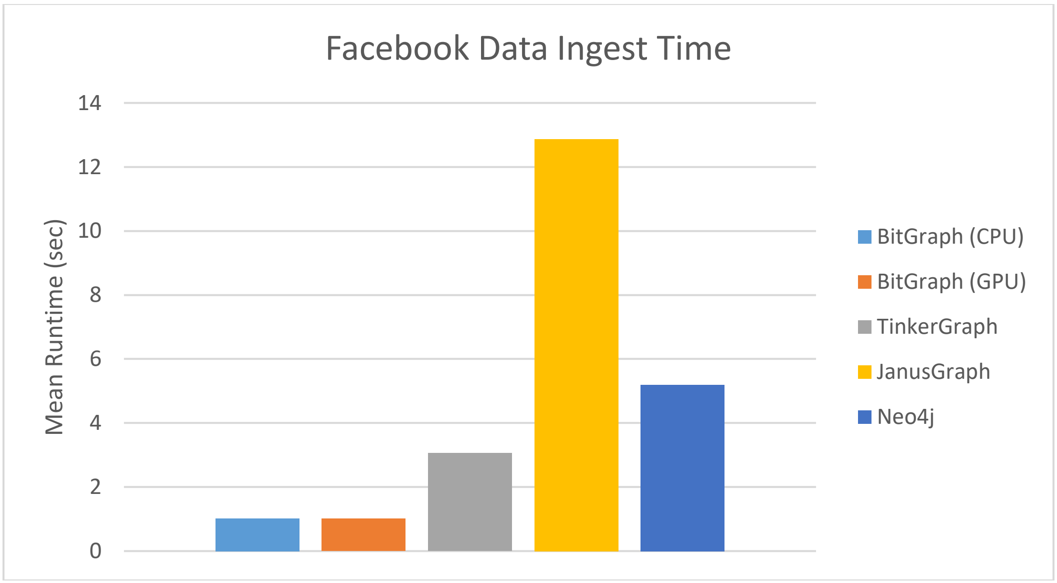


Figure 8: Facebook Data Ingest Time Comparison

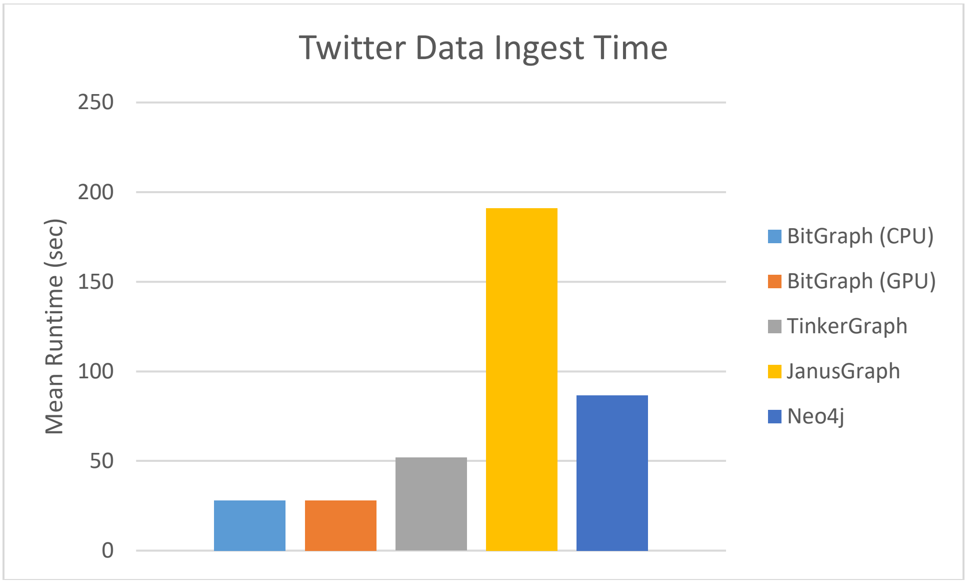


Figure 9: Twitter Data Ingest Time Comparison

Analysis

BitGraph stands out as the leader in data ingest, both on the CPU and GPU. It delivers roughly 2x speedup over its closest competitor, TinkerGraph, which is a mature product. Two key reasons for this include the performance of BitGraph's indexes and BitGraph's reliance on C++ vectors for bookkeeping. BitGraph's indexes are built from the ground up and designed to be graph indexes, unlike those that rely on higher-level constructs. This by itself offers significant speedup. The use of vectors for bookkeeping was not an initial design decision, but after profiling with gprof, I discovered that storing the BitVertex table in a dynamic array was up to 10x faster than using a linked list. The dynamic array implementation of choice, std::vector, is highly optimized and delivers impressive performance.

Connected Components

Results

The results of the cc1x algorithm (Table 5) for each backend on the ego-Facebook and ego-Twitter datasets are in Figure 10 and Figure 11. Because it is wasteful to use the GPU when processing small amounts of data, BitGraph-GPU allows a limit on the number of traversers that trigger the GPU Min Step, falling back to the CPU when this limit is not met. On the ego-Facebook data, this limit was set to 250 traversers; on the ego-Twitter data, this limit was set to 1000 traversers. These limits were chosen to ensure the GPU executed only on the most strenuous computations for each dataset. Without the limit, the slowdown due to data transfer dominated computation time, resulting a slowdown by at least one order of magnitude. Appendix B contains the entire table of results.

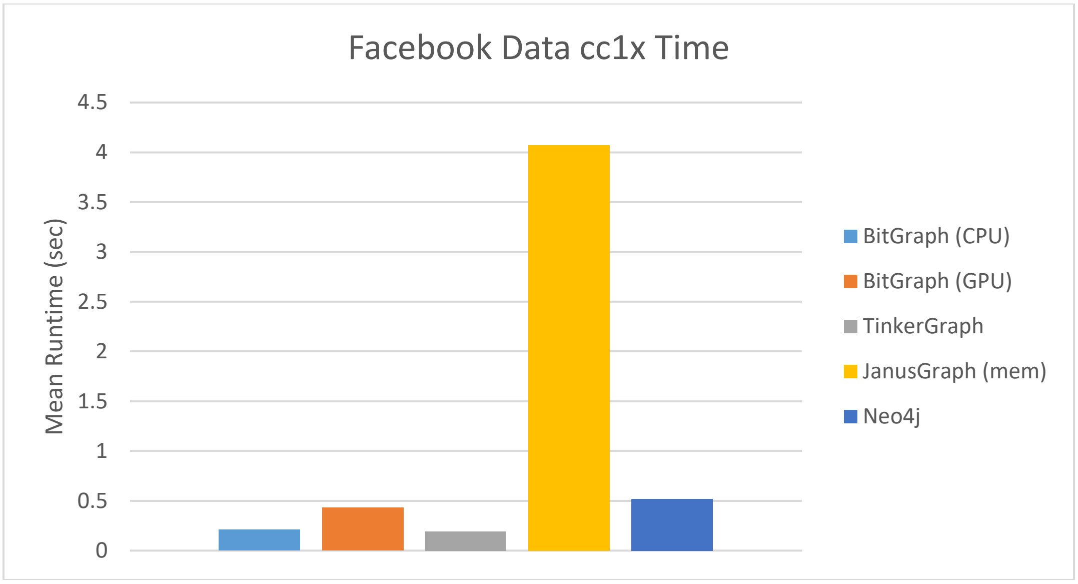


Figure 10: Facebook Data cc1x Time Comparison

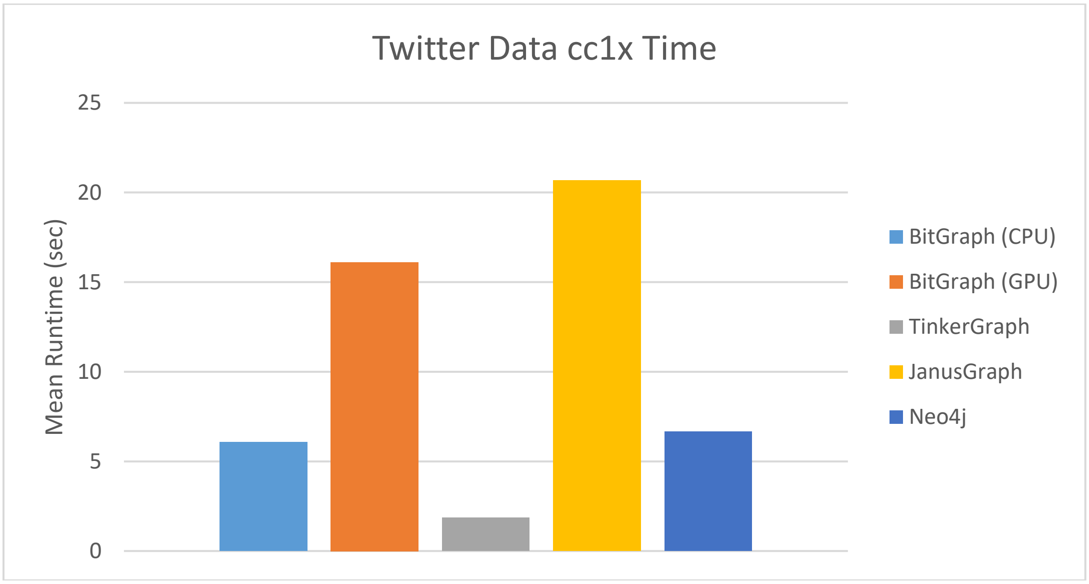


Figure 11: Twitter Data cc1x Time Comparison

Analysis

BitGraph did not perform as well on the cc1x traversal, losing to both TinkerGraph and Neo4j. TinkerGraph was particularly fast, completing the traversal at the same speed as BitGraph-CPU on the ego-Facebook data and 4x the speed of BitGraph-CPU on the ego-Twitter data. BitGraph-GPU did not perform well at all on either dataset, losing out to the CPU, JanusGraph, and Neo4j. The cc1x traversal was the best test of GPU acceleration, utilizing the Min Step on the GPU. However, it did not deliver speedup as the sparsity of the graph resulted in small vectors being sent to the GPU; these small vectors weren't large enough to make the data transfer time worth it. These results do not rule out the GPU as a viable acceleration option, but more work needs to be done to find the right use case. A different algorithm using the Has Step and Filter Step might be a good future test of the GPU's capabilities.

Degree Centrality

Results

The results of the degree centrality algorithm (Table 6) for each backend on the ego-Facebook and ego-Twitter datasets are shown in Figure 12 and Figure 13. Appendix C contains the entire table of results.

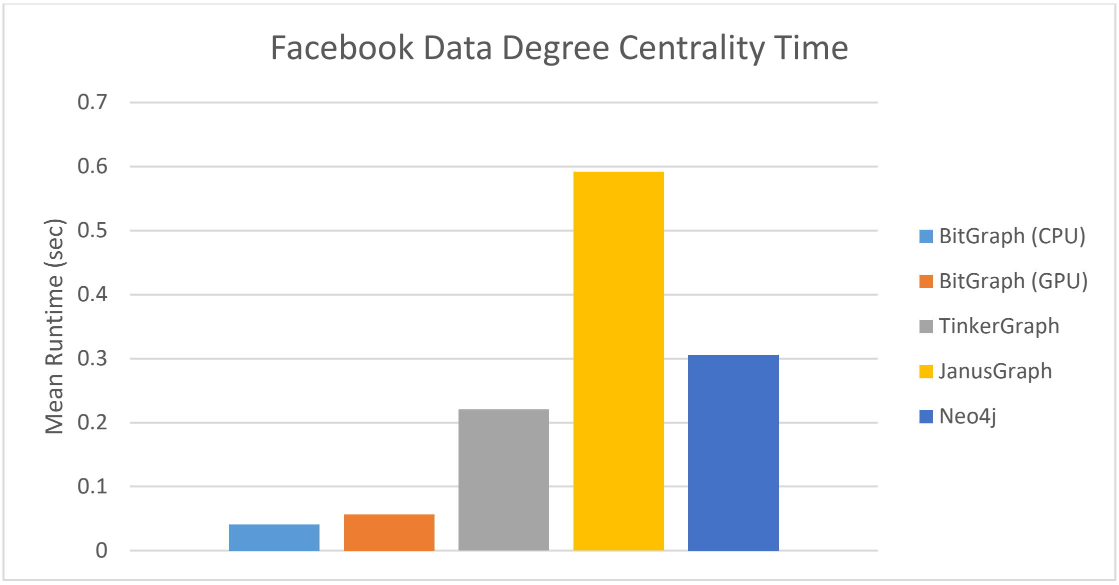


Figure 12: Facebook Data Degree Centrality Time Comparison

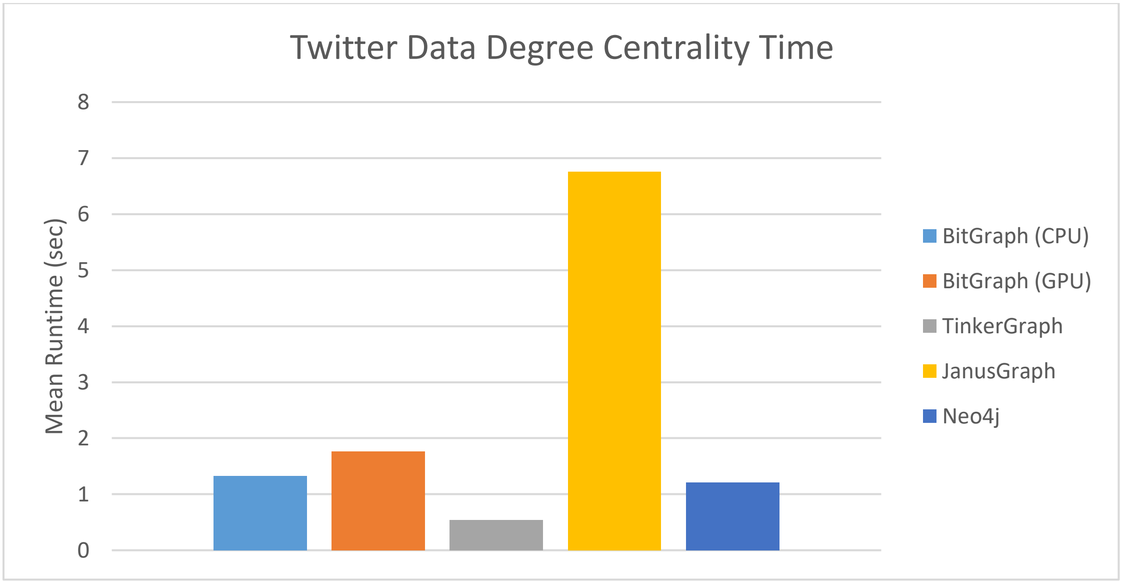


Figure 13: Twitter Data Degree Centrality Time Comparison

Analysis

Degree Centrality did not test the GPU (it does not use any GPU steps). As a result, BitGraph-GPU performance is nearly identical to that of BitGraph-CPU. On the ego-Facebook data, BitGraph outperformed all backends, delivering over 5x speedup over TinkerGraph, the second-fastest backend. However, on the ego-Twitter data, TinkerGraph once again performed the best, followed by Neo4j, then BitGraph-CPU.

Chapter 6: Conclusions & Future Work

Conclusions

Strengths

BitGraph is a powerful tool for data ingest, offering the fastest load times of any current Gremlin backend. While it does not offer the same speedup on connected components or degree centrality, it still has the fastest overall runtime due to the time saved in the data ingest step. Needless to say, this is a huge win for BitGraph, especially since it is not yet mature.

Limitations

At this time, BitGraph does not offer scalable speedup on connected components or degree centrality. This is likely because Java-Gremlin is able to extract parallelism from the traversal using Java streams, something which BitGraph currently does not do on CPU traversals. I experimented using OpenMP with BitGraph, but the traversals proved too complicated for OpenMP to parallelize. In the future, I will certainly explore parallelism on the CPU, as well as better use of the GPU.

Future Work

Overview

There are many potential improvements for Gremlin++ and BitGraph, including adding support for more traversal steps and simplifying how Gremlin++ handles predicates under the hood once OpenCL supports C++17. The clearest need is to implement standalone traversal strategies, which would greatly simplify the interpreter and make it easier to add new Gremlin++ supporting backends. Gremlin++ also needs support for common property graph file types such as graphml and graphson, which Java-Gremlin currently offers.

Traversal Steps

Today, BitGraph implements a small fraction of the many traversal steps of Gremlin. While many are equivalent, they do a great deal to make the language more expressive and convenient for users. One of the most important steps currently not implemented by BitGraph is the Repeat Step, which allows repetition of traversal elements without needing a new traversal. This would significantly the just-in-time compilation overhead when executing repeating traversals. Other priority steps would be the Dedup Step, a good target for GPU execution, and the Path Step, which facilitates an entire world of pathfinding algorithms, such as Betweenness Centrality. When OpenCL supports C++17, I also want to refactor the Has Step and Filter Step to make their predicates more like those in Java-Gremlin.

Better Traversers

At this time, BitGraph does not store any path or side effect information in its traversers, largely because Gremlin++ does not yet support those features. These are necessary for the Path Step, as well as other steps that refer backwards. When OpenCL supports C++17, I might also revisit the idea of templated traversers, which are type-specific rather than generic objects relying on boost::any.

Standalone Traversal Strategies

Traversal strategies in Gremlin++ and BitGraph are largely ad-hoc at the moment. In Java-Gremlin, backends can register strategies with the Java-Gremlin interpreter. These strategies handle backend-specific optimizations and any needed access to data in the backend. Gremlin++ only provides an interface method, *getInitialTraversal()*, which each backend implements to modify the user-supplied code as needed. Each backend also has to handle most of the interpretation, which is a barrier to developing new backends. In the future, I intend to fix these issues by allowing some form of standalone traversal strategies, which the backend passes to Gremlin++, instructing it on how to run backend-specific steps.

Data I/O

Java-Gremlin supports two common graph data formats, *graphml* and *graphson*. It also supports its own data format, *gryo*. At some point, Gremlin++ should also start to support these formats to facilitate usage with other graph systems and graph visualization engines.

Summary

Gremlin++, BitGraph, and this thesis represent a major step forward for the Gremlin language and the graph analytics community. As the first low-level language implementations of Gremlin and a Gremlin backend, they have encountered many previously-unknown pitfalls and bumps in the road to achieving a viable product. BitGraph has delivered on its promised speedup, but more work needs to be done in making it usable and extensible.

Appendices

Appendix A: Data Ingest Results

FACEBOOK RESULTS

	BitGraph (CPU)	BitGraph (GPU)	TinkerGraph	JanusGraph	Neo4j
Run 1	1.02046	1.01329	3.117	12.548	6.122
Run 2	1.04571	1.01084	3.141	12.656	4.931
Run 3	1.01208	1.01799	3.063	12.613	5.04
Run 4	1.02889	1.02009	3	13.874	5.156
Run 5	1.0102	1.01592	3.128	11.891	5.203
Run 6	1.0262	1.01722	3.031	13.149	5.04
Run 7	1.0087	1.01189	3.078	13.509	5.283
Run 8	1.00785	1.02531	3.055	12.14	5.153
Run 9	1.02033	1.01223	3.001	13.968	4.956
Run 10	1.01085	1.02026	3.031	12.319	5.017
Mean	1.019127	1.016504	3.0645	12.8667	5.1901

TWITTER RESULTS

	BitGraph (CPU)	BitGraph (GPU)	TinkerGraph	JanusGraph	Neo4j
Run 1	27.8411	27.8502	50.248	189.165	87.426
Run 2	27.9611	27.869	51.123	188.391	87.227
Run 3	27.919	28.0246	52.706	195.332	86.83
Run 4	27.966	27.9818	52.115	189.071	87.975
Run 5	28.0027	28.0038	52.727	195.307	88.593
Run 6	27.9773	28.004	52.414	191.043	87.72
Run 7	27.9754	27.9476	51.776	191.667	86.16
Run 8	27.6131	27.8146	51.359	192.735	86.14
Run 9	27.7475	27.7415	52.465	186.516	84.614
Run 10	27.9323	27.9127	52.324	190.672	84.569
Mean	27.89355	27.91498	51.9257	190.9899	86.7254

Appendix B: Connected Components Results

FACEBOOK RESULTS

	BitGraph (CPU)	**BitGraph (GPU)**	**TinkerGraph**	**JanusGraph**	**Neo4j**
Run 1	0.2081	0.436067	0.203	4.16	0.532
Run 2	0.20953	0.429677	0.223	3.914	0.484
Run 3	0.206442	0.428253	0.187	4.026	0.562
Run 4	0.205557	0.429988	0.187	4.03	0.59
Run 5	0.208029	0.43358	0.203	4.04	0.479
Run 6	0.209636	0.431179	0.203	3.9	0.5
Run 7	0.215564	0.4323	0.203	4.17	0.534
Run 8	0.209926	0.43604	0.187	4.17	0.53
Run 9	0.206598	0.432103	0.187	4.3	0.469
Run 10	0.209045	0.432801	0.172	4	0.513
Mean	0.2088427	0.4321988	0.1955	4.071	0.5193

TWITTER RESULTS

	BitGraph (CPU)	**BitGraph (GPU)**	**TinkerGraph**	**JanusGraph**	**Neo4j**
Run 1	6.1655	16.1637	1.963	17.476	6.971
Run 2	6.05749	16.1726	1.795	17.131	6.715
Run 3	6.03707	16.0375	1.795	21	6.746
Run 4	6.02782	16.0978	1.983	22.424	6.38
Run 5	6.10379	16.118	1.983	22.724	6.683
Run 6	6.08907	16.1091	1.805	20.266	7.428
Run 7	6.09618	16.1055	1.775	21.088	6.108
Run 8	6.11078	16.036	1.789	22.269	6.747
Run 9	6.0098	16.026	1.802	20.943	6.465
Run 10	6.04139	16.1673	1.773	21.277	6.372
Mean	6.073889	16.10335	1.8463	20.6598	6.6615

Appendix C: Degree Centrality Results

FACEBOOK RESULTS

	BitGraph (CPU)	**BitGraph (GPU)**	**TinkerGraph**	**JanusGraph**	**Neo4j**
Run 1	0.0408664	0.0573766	0.217	0.586	0.342
Run 2	0.0414136	0.0565604	0.223	0.613	0.288
Run 3	0.0410476	0.056404	0.223	0.577	0.322
Run 4	0.0411623	0.0564312	0.225	0.59	0.307
Run 5	0.0413689	0.0566073	0.221	0.584	0.292
Run 6	0.0416842	0.0577357	0.225	0.581	0.308
Run 7	0.0414839	0.0569546	0.222	0.585	0.302
Run 8	0.0414839	0.0575295	0.221	0.598	0.288
Run 9	0.0416105	0.0552968	0.212	0.593	0.302
Run 10	0.0412464	0.0566469	0.214	0.606	0.306
Mean	0.04133677	0.0567543	0.2203	0.5913	0.3057

TWITTER RESULTS

	BitGraph (CPU)	**BitGraph (GPU)**	**TinkerGraph**	**JanusGraph**	**Neo4j**
Run 1	1.32073	1.78519	0.505	7.635	1.128
Run 2	1.31869	1.75379	0.512	6.46	1.38
Run 3	1.32263	1.75355	0.556	6.576	1.196
Run 4	1.32174	1.74542	0.675	6.098	1.129
Run 5	1.35315	1.74807	0.511	6.995	1.124
Run 6	1.32639	1.74853	0.516	6.566	1.122
Run 7	1.31124	1.75508	0.542	8.083	1.117
Run 8	1.32842	1.75964	0.521	6.219	1.146
Run 9	1.3124	1.83258	0.558	6.862	1.329
Run 10	1.30031	1.79979	0.518	6.11	1.406
Mean	1.32157	1.768164	0.5414	6.7604	1.2077

Bibliography

[1] M. A. Rodriguez, "The Gremlin Graph Traversal Machine and Language," in *ACM Proceedings of the 15th Symposium on Database Programming Languages*, Pittsburgh, PA, USA, 2015.

[2] Apache Foundation, "Appendix: TinkerPop 2.x," 9 March 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/upgrade/#_tinkerpop_2_x.

[3] Neo4j Incorporated, "Neo4j," Neo4j Incorporated, 2019. [Online]. Available: https://neo4j.com. [Accessed 9 March 2019].

[4] Amazon Web Services, "Amazon Neptune: Fast, reliable graph database built for the cloud," Amazon Web Services, 2019. [Online]. Available: https://aws.amazon.com/neptune/. [Accessed 9 March 2019].

[5] The JanusGraph Authors, "JanusGraph: Distributed Graph Database," The Linux Foundation, 2017. [Online]. Available: http://janusgraph.org/. [Accessed 9 March 2019].

[6] Apache Foundation, "TinkerGraph-Gremlin," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/3.4.0/reference/#tinkergraph-gremlin. [Accessed 9 March 2019].

[7] Apache Foundation, "Gremlin Server," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/3.4.0/reference/#gremlin-server. [Accessed 9 March 2019].

[8] NVIDIA Corporation, "About CUDA," 2019. [Online]. Available: https://developer.nvidia.com/about-cuda. [Accessed 9 March 2019].

[9] Khronos Group, "OpenCL Overview," 2019. [Online]. Available: https://www.khronos.org/opencl/. [Accessed 9 March 2019].

[10] K. Henney, "Chapter 4. Boost.Any," 2001. [Online]. Available: https://www.boost.org/doc/libs/1_69_0/doc/html/any.html.

[11] Apache Foundation, "Provider Documentation," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/3.4.0/dev/provider/. [Accessed 24 March 2019].

[12] Apache Foundation, "The Traversal," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/3.4.0/reference/#traversal. [Accessed 9 March 2019].

[13] Boost Project, "Boost C++ Libraries," 12 December 2018. [Online]. Available: https://www.boost.org.

[14] J. Bay and A. Ruiz, "An Approach to Internal Domain-Specific Languages in Java," 19 February 2008. [Online]. Available: https://www.infoq.com/articles/internal-dsls-java. [Accessed 24 March 2019].

[15] Apache Foundation, "Has Step," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#has-step. [Accessed 24 March 2019].

[16] Apache Foundation, "A Note on Predicates," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#a-note-on-predicates. [Accessed 24 March 2019].

[17] Apache Foundation, "Add Property Step," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#addproperty-step. [Accessed 24 March 2019].

[18] Apache Foundation, "Vertex Properties," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/3.4.0/reference/#vertex-properties.

[19] Apache Foundation, "Properties Step," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#_properties_step. [Accessed 24 March 2019].

[20] Apache Foundation, "Graph Step," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#graph-step. [Accessed 24 March 2019].

[21] Apache Foundation, "Add Edge Step," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#addedge-step. [Accessed 24 March 2019].

[22] Apache Foundation, "Graph Computing," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#graph-computing. [Accessed 24 March 2019].

[23] Apache Foundation, "TinkerGraph-Gremlin," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin. [Accessed 24 March 2019].

[24] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Communications of the ACM,* vol. 16, no. 6, pp. 372-378, 1973.

[25] Apache Foundation, "ConnectedComponent Step," 2 January 2019. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#connectedcomponent-step. [Accessed 24 March 2019].

[26] J. Bisht, "Degree Centrality (Centrality Measure)," 2019. [Online]. Available: https://www.geeksforgeeks.org/degree-centrality-centrality-measure/. [Accessed 24 March 2019].

[27] J. Leskovec and R. Sosi, "SNAP: A General-Purpose Network Analysis and Graph-Mining Library," *ACM Transactions on Intelligent Systems and Technology (TIST),* vol. 8, no. 1, p. 1, 2016.

[28] P. Shannon, A. Markiel, O. Ozier, N. Baliga, J. Wang, D. Ramage, N. Amin, B. Schwikowski and T. Ideker, "Cytoscape: a software environment for integrated models of biomolecular interaction networks," *Genome Research,* pp. 2498-2504, 2003.

This page intentionally left blank.