A Roadmap for the GraphBLAS C++ API

Benjamin Brock*, Aydn Buluć†*, Timothy G. Mattson† Scott McMillan§ and José E. Moreira¶
* EECS Department, University of California, Berkeley, CA
† Computational Research Department, Lawrence Berkeley National Laboratory, Berkeley, CA
‡ Parallel Computing Labs, Intel, Hillsboro, OR
§ Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA
¶ IBM Thomas J. Watson Research Center, Yorktown Heights, NY

Abstract—The GraphBLAS is an API for graph algorithms expressed in terms of linear algebra. The current GraphBLAS specification is for the C Programming Language. Implementations of the GraphBLAS exposed a number of limitations due to C that restrict both the expressiveness and the performance of the GraphBLAS. The C++ language’s first-class support for generics, including template metaprogramming, addresses these limitations, yielding a simpler GraphBLAS API that should deliver better performance especially for methods based on user-defined types and operators. When combined with the pervasiveness of C++ across many domains as well as within large-scale distributed codes, we see a compelling argument to define a GraphBLAS C++ API. This paper presents a roadmap for the development of a GraphBLAS C++ API with a focus on the open issues we must resolve before completing the specification. Our goal is to foster discussion within the GraphBLAS user community and receive feedback on the directions we are taking with the GraphBLAS C++ API.

I. INTRODUCTION

The GraphBLAS Forum was formed in 2013 to standardize the mathematics and application programming interfaces (APIs) for performing graph computations in the language of linear algebra [10]. The mathematical specification was published in 2016 [9], followed one year later by the first version of the GraphBLAS C API Specification [3]. With input from early implementations from IBM [6] and Texas A&M University [5], we have refined the specification which is currently at version 1.3.

The C language was chosen for the first GraphBLAS API due to its ubiquity. C is supported on all mature hardware, and for research systems it is often the only language available. C++, however, has never been far from our consideration. Two C++ projects that predated the GraphBLAS, the Combinatorial BLAS [2] (CombinBLAS) and the GraphBLAS Template Library (GBTL) [11, 17], were influential and greatly informed the early development of the C API.

We believe it is critical that we develop a C++ language binding to the GraphBLAS for the following reasons:

1) Expressiveness: C++ provides an expressive interface with well-defined object lifetimes, constructors, destructors, and strong generic support through templates that greatly simplify support for user-defined types and operators.

2) Performance: Templates, combined with operator overloading, functors, and other callable types, enables inlining and powerful compiler code optimization to achieve more efficient support for GraphBLAS user-defined types and operators.

3) Pervasiveness: C++ use is growing among performance-oriented programmers, especially for new coding projects. Unfortunately, it is not straightforward to expose a C++ API through a C library, which necessitates a new API specialized to the needs of C++.

The primary contribution of this paper is the description of our plans for the GraphBLAS C++ API to support discussions about how we can best meet the needs of C++ programmers with the GraphBLAS.

II. LIMITATIONS OF THE C API

With three years of implementation experience, we now understand many limitations in the GraphBLAS C API. They come about primarily due to C’s lack of support for templates/generics. Three topics best expose these issues:

1. Limitations due to user-defined types The C API lists predefined support for the eleven Plain Old Data (POD) types. Additional types can be user-defined but these must be trivially copyable types, meaning they can be copied byte-for-byte, as with memcpy. This restriction reduces the API’s complexity and improves performance by eliminating the need for a function call whenever an object with a user-defined type must be copied. However, the literature on graph algorithms expressed as linear algebra has shown numerous cases where simple scalar types are not enough. For example, it is often necessary to use structs/classes with non-default constructors or contain complex objects involving memory dynamically allocated through pointers. C++ generics overcome these limitations by allowing types with custom copy constructors. This allows for more complex user-defined types, including C++ Standard Template Library containers such as std::vector and those that require other types of internal resource management.

2. Performance issues with user-defined operators Any operator that is not predefined by the C API is a user-defined operator. These could consist of: (a) operations that are not supported over the predefined types, (b) operations that operate on user-defined types, or (c) a combination of (a) and (b). In C, these user-defined operators must be implemented as function pointers that take and return void*. For example:

\[
\text{void \text{\texttt{(}} \text{unary\_func\text{)(void *out, const void *in)};} \text{\texttt{)}} \]

3. Well-defined object lifetimes, constructors, destructors, and strong generic support through templates that greatly simplify support for user-defined types and operators.
In addition to the verbose syntax and lack of type safety inherent in C’s function pointer interface, using C function pointers for user-defined operators results in severe performance degradation. This is because every call to this operator will result in an indirect function call, preventing compiler inlining and subsequent code optimizations.

3. Code bloat due to predefined types and operators The C API defines dozens of unary and binary operators, monoids and semirings; each defined for up to eleven different predefined types. Since C lacks support for function overloading and generics, GraphBLAS C library implementers typically resort to automatic code generation to produce a very large set of functions for each type and operator combination. Although this approach has been used successfully in the SuiteSparse GraphBLAS [5], the additional complexity is daunting for many hopeful GraphBLAS library developers.

C++ templates resolve many of these issues. C++ user-defined types improve inlining and optimization opportunities compared to C function pointers. C++ eliminates runtime costs associated with indirect function calls for user-defined types. In addition, C++ supports more expressive user-defined types with a type’s behavior described through copy constructors, move constructors, and other lifetime events. C++ allows function overloading so the same function name can be used for invocations with different types. This simplifies the API while reducing the work of library implementors since they can use templates to avoid writing GraphBLAS methods for each type. Finally, C++ overloading allows vendors to provide custom, hand-tuned versions of some GraphBLAS methods with fixed types for increased performance.

III. RELATED WORK

Numerous C++ linear algebra frameworks exist to help guide our work on the C++ GraphBLAS API. C++ graph libraries that closely mimic the GraphBLAS C API include the GraphBLAS Template Library [1], [17], GraphBLAST [16], and IBM GPI’s C++ API [6]. Another C++ library for graph systems based on linear algebra includes GraphMat [15].

General C++ linear algebra frameworks of note include Eigen, a high-level matrix library for C++ [2]. Eigen provides high-level, generic data structures for matrices and vectors (both dense and sparse), along with a collection of algorithms. Eigen uses expression templates to provide code-specific compile-time optimizations. This means that certain operations return expression objects instead of explicitly materialized intermediate products, allowing for operator fusion, loop unrolling, and vectorization. The Tensor Algebra Compiler (TACO) is a C++ DSL for sparse tensor computations that allows users to optimize computational kernels on sparse tensors [11]. There are multiple proposals for extensions to the C++ Standard Library for linear algebra container types and operations [4], [8]. Finally, there is a proposed extension to the C++ Standard Library for graphs [14].

In the distributed computing space, the Combinatorial BLAS provides many of the necessary sparse matrix primitives necessary for the GraphBLAS in a C++ library built on top of MPI [2]. Another similar approach is found in Elemental [13].

IV. KEY FEATURES OF A GRAPHBLAS C++ API

The C++ language has a number of features that will influence the design of the GraphBLAS C++ API, including generics (through templates, template parameter deduction and metaprogramming), constructors and destructors, function overloading, operator overloading, exceptions, and concepts (in C++20). The C++ Standard Library also includes a broad array of software utilities, such as containers, pre-defined mathematical operators, and tools for type inspection, that support interoperability and influence our design of the GraphBLAS C++ API. The language is constantly evolving with a new release approximately every three years; therefore tracking proposed changes to C++ will be part of this development process. In this section, we highlight some of the features that will directly impact the GraphBLAS C++ API.

A. Namespaces and Scoping

The C API used the prefix GrB_ to identify all API elements. In the C++ API, the grb namespace will be used to scope everything within the API. Nested namespaces within grb (such as detail) will be used to scope non-API, and implementation-defined portions of the library. A valid GraphBLAS program will only access elements from the root grb namespace.

B. Domains

The C API used symbols to denote the predefined domains (i.e., types) that could be stored in matrices and vectors (e.g., GrB_BOOL, GrB_INT8, GrB_FP32, etc.). Using C++ template mechanisms, these symbols are not needed and types (including user-defined) will be directly specified (e.g., bool, int8_t, float, etc.) as template parameters.

C. Error Handling and Exceptions

An important open issue is how to handle errors. The GraphBLAS C API handles errors through special error values returned from every GraphBLAS method. Returning errors this way has long been held to be the most performant method for handling errors in C++ (especially since early implementations of exceptions incurred significant performance overheads). More recent implementations of exceptions, however, result in minimal performance overhead relative to traditional error return value checking for many use cases.

Furthermore, in modern C++ best practices, exceptions are the preferred way to handle errors because they force the calling code to recognize and deal with errors (otherwise program execution stops), the errors can be handled at any level of the call stack, and in the process of unwinding the stack, objects are properly destructed based on well-defined rules. In addition, exceptions foster cleaner code by clearly separating code that handles errors, and exceptions also allow API functions to return values more naturally. Therefore, the GraphBLAS C++ API will consider using C++ exceptions for interface usability, correctness, and performance reasons.
D. Containers: Vectors and Matrices

In the GraphBLAS C API, vectors and matrices are opaque objects meaning that information about the storage of data is unknown to the user of the object. This is analogous to the information hiding concept of classes in C++ where the user may interact with an object only through the class’s public interface, while all implementation details are hidden in the private section of the class. We will consider the same approach in the C++ specification. We used this approach in GBTL as illustrated in Figure 1 for matrices.

A template parameter specifies the scalar type (domain) stored in a matrix or vector. Other template parameters might allow the user to specify the index type (currently restricted to uint64_t in C API), and provide additional traits such as sparse vs. dense, directed vs. undirected, etc. In addition, all C API methods for building, modifying and querying vectors or matrices would be public members of the vector/matrix class. Figure 1 shows a Matrix constructor that would have the same functionality as GrB_Matrix_new() and a method for querying the number of rows (replacing GrB_Matrix_nrows()). All methods in this API class would forward calls to the “backend” Matrix class that is defined by an implementation.

E. GraphBLAS Operators

GraphBLAS C API Version 1.3.0 provides a set of over 50 predefined operators – unary and binary functions, monoids and semirings – that can be used within GraphBLAS programs. Each is defined for a set of built-in domains. For example, to use the plus operator with objects of type int32_t, the user must select GrB_PLUS_INT32. Through the use of template structs (functors), C++ offers a more elegant mechanism for supporting operators that execute on different types. An example of the Plus operator from GBTL is shown in Figure 2. This is more versatile than the eleven variants defined in the C API since this template lets the inputs and the output be of different types (even non-POD types).

Another idea for the C++ API is support for a collection of generic, pre-defined operators found in the Standard Template Library (STL) <functional> header, such as std::plus and std::multiplies. These are defined as template structs. They can handle arguments of various types while enabling aggressive compiler inlining for increased performance. In a future GraphBLAS C++ API, we foresee a series of GraphBLAS C++ operators that follow the standard library’s conventions while supporting the use of the STL’s operators.

A more complex open issue relating to GraphBLAS operators in C++ is stateful operators. These are operators which might carry an internal state used to determine the output of operations. A key point is whether stateful operators should be allowed, and, if so, what the semantics should be in terms of copying or borrowing the operators, and whether the order in which the operator is invoked may vary between executions.

F. GraphBLAS Operations

The GraphBLAS C++ API will feature versions of the GraphBLAS operations, such as mxm and mxv, that closely mimic the C API functions with names and arguments mainly in the same. An example C++ mxv operation taken from GBTL is shown in Figure 3 where the first six arguments are equivalent to the ones in the C API. The only difference is the last argument where the C API would have a Descriptor. This difference is briefly discussed in the next section. Also note that (except for the last) the type of each argument is completely templated to allow for the complete range of possible types. A concurrent effort to extend GraphBLAS to distributed environments considers introducing a GraphBLAS “context” that would be passed into each GraphBLAS operation, potentially unifying serial, multi-threaded, and distributed APIs. It is convenient to have GraphBLAS operations be the methods of a future GraphBLAS context object.

Another idea under consideration is to mimic syntax from high-level libraries such as NumPy, where matrix-vector multiplication could also be performed as methods embedded in container classes (\(w = A\cdot mxv(u)\)). This would require that the output parameter be returned from the method. This would be more concise and would provide a more friendly programming...
environment for novice users. On the other hand, this goes against C++ best practices, in which algorithms (operations in this case) are separate from, but interoperate with containers.

It will be necessary to carefully weigh the convenience and conciseness of a user API against the implementer burden required to support it.

G. GraphBLAS Descriptors and Views

As shown in Figure 3, the place where GBTL deviates most from the C API is the use of Descriptors. In the C API, Descriptors control whether or not input matrices are transposed, masks are complemented, or if unmasked elements of the output are cleared (the replace flag). For GBTL, a design decision was made to improve readability and allow the template system to generate more efficient code. Compare the following two mxv calls:

```
1 mxv(w, m, GrB_PLUS_INT32, GrB_PLUS_TIMES_SEMIRING_INT32,
   A, u, GrB_Desc_RCT0);  // C API
2 mxv(w, complement(m), Plus<int>, PlusTimesSemiring<int>,
   transpose(A), u, REPLACE);  // GBTL C++ API
```

Instead of encoding “complement the mask, transpose the first input matrix, and set the replace flag” in a descriptor at the end of the argument list, GBTL uses complement views to wrap masks, transpose views to wrap input matrices and a replace flag. In the C++ code above, the intent is clearer at the call site. Through generics, these views are types that can be used to select which code paths are most efficient without necessarily having to materialize complements or transposes.

Another use of views is for indexing and performing operations on submatrices without explicitly copying the matrix. For a future GraphBLAS C++ API, it is important to consider user demand and implementer effort required to support various high-level views. Much more discussion is needed before these issues are settled in the C++ API.

V. CONCLUSIONS AND FUTURE WORK

The GraphBLAS C API is well established. A growing community is developing algorithms and contributing them to the LAGraph project [12]. We see an increased demand for a C++ API, especially from large scale, parallel and distributed framework developers. There are C++ libraries that already implement all or most of the mathematical functionality of GraphBLAS [1, 2, 16] on which we can base this effort. In addition, a number of proposed C++ Standard Library extensions in the linear algebra [4, 8] and graph domains [14] will influence our work. We also anticipate working on the C++ API in conjunction with work on distributed and parallel support. Hence, we are carefully considering the parallel execution policy extensions introduced in C++17 and how they may influence our design.

ACKNOWLEDGMENTS AND DISCLAIMERS

We thank the members of the GraphBLAS forum. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM20-xxxx]. Benjamin Brock and Aydin Buluç were supported in part by the DOE Office of Advanced Scientific Computing Research under contract number DEAC02-05CH11231 and in part by NSF under Award No. 1823034.

REFERENCES