A Roadmap for the GraphBLAS C++ API

Benjamin Brock*, Aydın Buluç†*, 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, Pittsburgh, PA
¶ IBM Thomas J. Watson Research Center, Yorktown Heights, NY

***Abstract*—The GraphBLAS are building blocks for expressing graph algorithms in terms of linear algebra. Currently, the GraphBLAS are defined as a C API. Implementations of the GraphBLAS have exposed limitations in expressiveness and performance due to limitations in C. A move to C++ should address many of these limitations while providing a simpler API. Furthermore, for methods based on user-defined types and operators, the performance should be significantly better. C++ has grown into a pervasive programming language across many domains. We see a compelling argument to define a GraphBLAS C++ API. This paper presents our roadmap for the development of a GraphBLAS C++ API. Open issues are highlighted with the goal of fostering discussion and generating feedback within the GraphBLAS user community to guide us as we develop the GraphBLAS C++ API.**

I. Introduction

The GraphBLAS Forum was formed in 2013 to define common building blocks 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] (CombBLAS) and the GraphBLAS Template Library (GBTL) [1], [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 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 to describe our plans for the GraphBLAS C++ API and 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 that 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:

```
1    void (*unary_func)(void *out, const void *in);
```

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

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00049

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 12,2022 at 21:35:28 UTC from IEEE Xplore. Restrictions apply.

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 resort to automatic code generation to produce a 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 inform our work. 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++ [7]. 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 tenors [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. This is the approach used by all of the C++ frameworks listed in Section III.

C. Error Handling and Exceptions

An important open issue is how to handle errors. The GraphBLAS C API handles errors through error-values returned from every GraphBLAS method. This approach has long been held as the most performant way to handle errors in C++ (especially since early implementations of exceptions incurred significant performance overheads). More recent implementations of exceptions, however, add minimal overhead relative to approaches based on error-values. Furthermore, current C++ best-practices view exceptions as the preferred way to handle errors. Exceptions force the calling code to recognize and deal with errors. Exceptions are 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. Finally, by using exceptions for error conditions, the return values from GraphBLAS methods can be used for more semantically meaningful purposes. A majority of the frameworks listed in Section III manage errors through exceptions and we anticipate doing so as well as we define the GraphBLAS C++ API.

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 hidden from users. This is analogous to classes in C++ where

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 12,2022 at 21:35:28 UTC from IEEE Xplore. Restrictions apply.

```
template<typename ScalarT, typename... OtherTagsT>
class Matrix
{
public: // the specified API appears here
  Matrix(IndexType rows, IndexType cols)
      : impl_mat(rows, cols) {}

  IndexType nrows() const { return impl_mat.nrows(); }

private:  // this section is implementation-defined
  // Not shown: implementation-defined type generator
  ImplementationMatrixType impl_mat;
};
```

Fig. 1. GBTL Matrix class.

```
template<typename D1, typename D2 = D1, typename D3 = D1>
struct Plus
{
  typedef D3 result_type;
  inline D3 operator()(D1 lhs, D2 rhs) {return lhs+rhs;}
};
```

Fig. 2. GBTL Plus operator.

users 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. This approach was used in GBTL as illustrated in Figure 1 for matrices.

A template parameter specifies the scalar type stored in a matrix or vector. Another template parameter could specify the index type (currently restricted to `uint64_t` in C API) as is done in CombBLAS, Eigen, and the proposed C++ Standard Graph Library. Additional template parameters could specify storage traits such as sparse vs. dense, directed vs. undirected, and so forth. In addition, methods from the C API 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()` from the C API 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 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. 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 over different types. Each of the C++ frameworks listed in Section III that support custom operators do so through callable functors. 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 allows the inputs and the output to be of different types (and even non-POD, user-defined types).

Another idea for the C++ API is support for the collection of generic, pre-defined function objects found in the Standard Template Library's (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.

A more complex open issue relating to GraphBLAS operators in C++ is *stateful* operators. These are operators which carry internal state used to determine their output. 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 similar names and arguments. An example C++ `mxv` operation taken from GBTL is shown in Figure IV-F 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. This implies a design in C++ where the GraphBLAS operations would be the methods of a future GraphBLAS context object.

Another option is to mimic syntax from high-level libraries such as NumPy, where matrix-vector multiplication would be performed as methods embedded in container classes (`w = A.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. However, this goes against C++ best practices, in which algorithms (operations in this case) are separate from, but interoperate with, containers.

```
template<typename WVectorT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename UVectorT>
inline void mxv(WVectorT         &w,     // output
                MaskT      const &mask,
                AccumT     const &accum,
                SemiringT  const &op,
                AMatrixT   const &A,
                UVectorT   const &u,
                OutpControl       outp); // MERGE/REPLACE
```

Fig. 3. GBTL C++ mxv signature.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 12,2022 at 21:35:28 UTC from IEEE Xplore. Restrictions apply.

G. GraphBLAS Descriptors and Views

As shown in Figure IV-F, 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, we decided to improve readability and allow the template system to generate more efficient code. Compare the following two mxv calls:

```
mxv(w, m, GrB_PLUS_INT32, GrB_PLUS_TIMES_SEMIRING_INT32,
    A, u, GrB_Desc_RCT0);              // C API
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.

Views can also be used for indexing and performing operations on submatrices without explicitly copying the matrix. Handling views in a way that meets user's needs without undue complexity for the implementor will be a key issue as we define 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 our work on a C++ API. We will also align our work closely around proposed C++ Standard Library extensions in the linear algebra [4], [8] and graph domains [14]. Work on the C++ API will occur in conjunction with work on distributed and parallel support. Hence, we are following 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 [DM20-0209]. Benjamin Brock and Aydın 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

[1] Graphblas template library (GBTL), v. 2.0. https://github.com/cmu-sei/gbtl.

[2] Aydın Buluç and John R Gilbert. The combinatorial blas: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[3] Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. The graphblas c api specification. *GraphBLAS. org, Tech. Rep.*, version 1.3.0, 2019.

[4] Guy Davidson and Bob Steagall. P1385r5: A proposal to add linear algebra support to the c++ standard library. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1385r5.pdf, January 2020.

[5] Timothy A Davis. Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[6] K. Ekanadham, W. P. Horn, Manoj Kumar, Joefon Jann, José Moreira, Pratap Pattnaik, Mauricio Serrano, Gabriel Tanase, and Hao Yu. Graph Programming Interface (GPI): A linear algebra programming model for large scale graph computations. In *Proc. ACM Intl. Conference on Computing Frontiers*, CF '16, pages 72–81, New York, NY, USA, 2016.

[7] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[8] Mark Hoemann, David Hollman, Christian Trott, Daniel Sunderland, Nevin Liber, Siva Rajamanickam, Li-Ta Lo, Graham Lopez, Peter Caday, Sarah Knepper, Piotr Luszczek, and Timothy Costa. P1673r1: A free function linear algebra interface based on the BLAS. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1673r1.html, June 2019.

[9] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[10] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[11] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[12] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. Lagraph: A community effort to collect graph algorithms built on top of the graphblas. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–284. IEEE, 2019.

[13] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2), February 2013.

[14] Phillip Ratzloff, Richard Dosselmann, and Michael Wong. P1709r1: Graph library. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1709r1.pdf, January 2019.

[15] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[16] Carl Yang, Aydın Buluç, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *arXiv preprint arXiv:1908.01407*, 2019.

[17] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 912–920. IEEE, 2016.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 12,2022 at 21:35:28 UTC from IEEE Xplore. Restrictions apply.