PyGB: GraphBLAS DSL in Python with Dynamic Compilation into Efficient C++

Jesse Chamberlin*, Marcin Zalewski†, Scott McMillan‡, and Andrew Lumsdaine*†
jessec18@uw.edu, marcin.zalewski@pnnl.gov, smcmillan@sei.cmu.edu, andrew.lumsdaine@pnnl.gov

*University of Washington
Seattle, Washington, USA

†Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
Seattle, Washington, USA

‡Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

***Abstract*—We present PyGB, a high-level Python domain-specific language (DSL) for GraphBLAS building blocks for graph algorithms. GraphBLAS is based on a small number of linear algebra operations, and it is described using mathematical notation and concepts. However, the concrete realizations of GraphBLAS concentrate on efficiency and widely usable programming interfaces. The GraphBLAS C language API standardized by the GraphBLAS Forum is the major example of such a realization, and our GraphBLAS Template Library (GBTL) implemented in the C++ language is another. PyGB exposes the C++ interface of GBTL through a DSL that leverages Python's mature scientific computing ecosystem and high-level syntax. The syntax of PyGB more closely resembles the GraphBLAS mathematical notation than GBTL. The DSL dispatches to dynamically compiled templated classes to achieve comparable performance to the native GBTL code. We highlight the features of PyGB through code examples and discuss how Python's syntax and dynamic execution enable us to provide the high-level abstraction with minimal performance penalty. We demonstrate the stages of execution, from type inference to template compilation, to dynamic linking and invocation from the Python interpreter. Our experimental evaluation shows that for sufficiently large inputs, the overhead of PyGB is negligible and compilation times are not worse than for native GBTL implementation. Last, we outline concrete features we plan to implement in the future to extend PyGB.**

***Index Terms*—GraphBLAS, HPC, C++, graph algorithms, Python, DSL**

I. Introduction

GraphBLAS [1], [2], [3], [4], [5], [6] is an emerging set of building blocks for graph algorithms, based on a small number of linear algebra operations. In the context of GraphBLAS, a graph is represented as an adjacency (or incidence) matrix, and operations on graphs correspond to linear algebra operations on the graph matrix. In contrast to BLAS (Basic Linear Algebra Subprograms) [7], which targets numerical computation, GraphBLAS parameterizes its operations with arbitrary semirings, allowing a wide range of computations on the edges of the graph. For example, Fig. 1 shows how a graph represented by an adjacency matrix can be multiplied by a vector representing a frontier of a breadth-first search (BFS) to obtain the next frontier using the semiring of Boolean algebra.

The GraphBLAS Forum [1], a diverse group with representatives across government, academia and industry, standardized a GraphBLAS API for the C programming language [5].

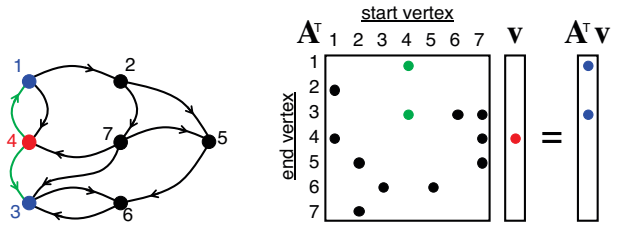

Fig. 1: One ply of BFS from source vertex 4 [4]

```
Input: graph, frontier, levels
depth ← 0
while nvals(frontier) > 0:
  depth ← depth + 1
  levels<frontier,merge> ← depth
  frontier<¬levels,replace> ← graph^T ⊕.⊗ frontier
    where ⊕.⊗ = ⊕.⊗(LogicalSemiring)
```

(a) Pseudocode

```
def bfs(graph, frontier, levels):
  depth = 0
  while frontier.nvals > 0:
    depth += 1
    levels[front][:] = depth
    with gb.LogicalSemiring, gb.Replace:
      frontier[~levels] = graph.T @ frontier
```

(b) PyGB

```
template<class Mat, class Frontier, class Levels>
void bfs(Mat& graph, Frontier frontier, Levels& levels)
{
  GB::IndexType depth = 0;
  while (frontier.nvals() > 0) {
    ++depth;
    GB::assign(levels, frontier, GB::NoAccumulate(),
               depth, GB::AllIndices(), false);
    GB::mxv(frontier, GB::complement(levels),
            GB::NoAccumulate(),
            GB::LogicalSemiring<GB::IndexType>(),
            GB::transpose(graph), frontier, true);
  }
}
```

(c) C++

Fig. 2: BFS in math pseudocode, PyGB, and C++

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00059

TABLE I: Comparison of GraphBLAS operations written in mathematical notation from C API [5] and PyGB notation

Operation Name	Mathematical Notation	PyGB Notation
mxm	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$	`C[M, z] = A @ B`
mxv	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$	`w[m, z] = A @ u`
eWiseMult	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$	`C[M, z] = A * B`
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$	`w[m, z] = u * v`
eWiseAdd	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$	`C[M, z] = A + B`
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$	`w[m, z] = u + v`
reduce (row)	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$	`w[m, z] = reduce(monoid, A)`
reduce (scalar)	$s = s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$	`s = reduce(A)`
	$s = s \odot [\oplus_i \mathbf{u}(i)]$	`s = reduce(u)`
apply	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot f_u(\mathbf{A})$	`C[M, z] = apply(A)`
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot f_u(\mathbf{u})$	`w[m, z] = apply(u)`
transpose	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A}^T$	`C[M, z] = A.T`
extract	$\mathbf{C}\langle\mathbf{M}, z\rangle = \mathbf{C} \odot \mathbf{A}(\boldsymbol{i}, \boldsymbol{j})$	`C[M, z] = A[i, j]`
	$\mathbf{w}\langle\mathbf{m}, z\rangle = \mathbf{w} \odot \mathbf{u}(\boldsymbol{i})$	`w[m, z][i, j] = u`
	$\mathbf{C}\langle\mathbf{M}, z\rangle(\boldsymbol{i}, \boldsymbol{j}) = \mathbf{C}(\boldsymbol{i}, \boldsymbol{j}) \odot \mathbf{A}$	`C[M, z][i, j] = A`
	$\mathbf{w}\langle\mathbf{m}, z\rangle(\boldsymbol{i}) = \mathbf{w}(\boldsymbol{i}) \odot \mathbf{u}$	`w[m, z][i] = u`

Currently, the SuiteSparse library is the only complete implementation of the C API. In addition, there are several projects developing GraphBLAS implementations [8], [9], [10], [11], [12], [13], including our C++ GraphBLAS Template Library (GBTL) [14], [15]. Despite the ongoing efforts, there is no easily accessible, high-performance DSL for GraphBLAS. The main contribution of this paper is such a GraphBLAS DSL, implemented in Python [16] and called PyGB (Python GraphBLAS)[1], that provides a high-level programming interface, and at the same time dynamically compiles to a highly-performing C++ implementation. Figure 2 shows the simplified BFS algorithm in math-based pseudocode based on the notation in C API Specification (Fig. 2a), the corresponding code in PyGB, and the C++ code in GBTL. PyGB provides a notation that is very close to the pseudocode, and, at the same time, it achieves the efficiency of the GBTL code to which it is compiled.

PyGB leverages the advantages of Python, which has a strong foothold in the scientific community. A mature ecosystem of existing matrix and graph libraries and tools is made available by Python. The pseudocode-like syntax and interpreted execution model make Python a popular language for rapid prototyping. Tools such as Jupyter [17] make it possible to prototype GraphBLAS code interactively from a web browser. Python is also well-suited for piping data between disparate libraries and tools. PyGB provides convenience constructors for copying data from NumPy arrays, SciPy sparse matrices, and NetworkX graphs. In the future (see Sec. VIII) we plan to further integrate PyGB with the Python HPC stack. Python provides a large set of overloadable operators, which we utilize to provide simple, expressive syntax that is as powerful as the more complex syntax of GBTL and the C API. Last, we make PyGB efficient and future proof by compiling operations in the Python DSL to GBTL code, taking a similar approach to other high-performance Python libraries that rely on lower-level languages for performance-critical sections of code.

[1]Repository at https://github.com/jessecoleman/gbtl-python-bindings.

Contributions:

- We provide PyGB, a high-level Python DSL for GraphBLAS that closely corresponds to pseudocode math-like notation. Our DSL relies on Python syntax features such as magic methods [18] and context managers [19] and integrates with NumPy, SciPy and NetworkX to deliver a user-friendly development experience. We give an example-driven overview of the DSL in Sec. III and we discuss design considerations in Sec. IV.
- PyGB is dynamically compiled to GBTL (see Sec. V). The dynamic compilation process keeps track of C++ types and templates that have been already compiled, and invokes the C++ compiler as necessary, dynamically binding the symbols in the binary files generated by the C++ compiler. While the current implementation compiles to GBTL, our method can be generalized to target any other library written in C or C++, making our DSL future proof by being able to draw upon developments in the field. Our methodology can also be used to develop other DSLs in Python.
- We show that the overhead of PyGB becomes negligible as the problem size scales. In Sec. VI, we present empirical evidence by evaluating several algorithms, comparing the run time of PyGB generated code where outer loops are performed in Python, PyGB code where outer loops are performed in C++, and native GBTL code.

II. GraphBLAS Mathematical Notation

The GraphBLAS C API Specification is based on the mathematical behavior of the GraphBLAS operations that was defined by Kepner, et al. [3]. The specification added

a number of extensions and provides a mathematical notation that captures this behavior. The C API Specification notation for many of the GraphBLAS operations can be found in column 1 of Table I. As in the math document, the operations denoted by $\oplus$ and $\otimes$ on the right-hand side are analogous to traditional arithmetic addition and multiplication from linear algebra, but can be replaced by any pair of semiring operators where the identity of $\oplus$ is the annihilator of $\otimes$. Although not shown in the table, input matrices $\mathbf{A}$ and $\mathbf{B}$ may be optionally transposed. An optional accumulate specified by $\odot$ allows the user to specify a separate binary operator that governs how the results of the operation are "added" to existing values in the output object. Fig. 4a has examples of both the transpose and accumulation operations in PyGB and GBTL for comparison.

The other major extension supported by the specification is the use of a write mask and replace flag. In the expression $\mathbf{C}\langle\mathbf{M}, z\rangle$ the output matrix, $\mathbf{C}$, is masked by a boolean matrix $\mathbf{M}$ of the same dimension whose values control which elements resulting from the operation are written to $\mathbf{C}$. The elements of the mask can be optionally complemented to invert the set of elements that can be written. When specified, the "replace" flag, z, indicates that all values in the output object are cleared ("zeroed") prior to the masked assignment. When the z flag is not specifed and a mask has been specified, "merge" behavior occurs where elements in the output matrix that are "masked out" retain their original contents.

Although there is only one official language specification at this time, it is intended that all compliant implementations of GraphBLAS adhere to the behavior defined by this notation regardless of the language. The goal of this Python DSL is to not only adhere to the behavior specified but also to replicate the mathematical notation. The latter will lessen the learning curve as the notation is similar to linear algebra. It also results in a simpler more compact notation than has been possible with the C API, which is characterized by function signatures with many parameters which bear little resemblance to the mathematical notation.

III. PyGB

The Matrix and Vector classes in PyGB contain mutable graph data which can be constructed both from sparse data in coordinate format and dense data in 2D array format, as in Fig. 3a. The user may optionally specify a data type to cast the values to. Containers can also be constructed from NumPy arrays, SciPy.sparse matrices, and NetworkX graphs as in Fig. 3b. This allows the user access to a wide range of well documented graph-generating routines. PyGB currently performs a data copy at construction, but methods are being explored to avoid this (see Sec. VIII).

Matrices and Vectors act as the operands to PyGB operation functions. Table I provides a comprehensive list of the operations defined by GraphBLAS and their equivalent PyGB syntax. At execution, these expressions are evaluated by machine code that is compiled on-demand based on operand data types. With all expensive matrix/vector operations optimized by the DSL, higher-level logic such as outer loops and control flow can be written in concise Python syntax with minimal overhead. We will demonstrate some of the more complex syntax of PyGB through code examples.

```python
# construction from sparse data
m = gb.Matrix((vals, (row_idx, col_idx)), shape=(r, c))
v = gb.Vector((vals, idx), shape=(l,))
# construction from dense data
m = gb.Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
v = gb.Vector([1, 2, 3, 4, 5])
```

(a) copy from Python lists

```python
# construction from NumPy random matrix
m = gb.Matrix(np.random.rand(3, 3))
# construction from SciPy sparse diagonal matrix
m = gb.Matrix(sc.sparse.diags([1, 1, 1], [-1, 0, 1],
        shape=(3,3)))
# construction from NetworkX balanced tree
m = gb.Matrix(nx.balanced_tree(r=4, h=8))
```

(b) copy from other library containers

Fig. 3: Constructing PyGB containers

The single-source shortest path algorithm (Fig. 4a) demonstrates the use of a semiring and accumulator used in conjunction. `path` is multiplied by `graph.T` (transpose) and then accumulated back into `path`. Inside the `with` block, a PyGB pre-defined semiring and an anonymous user-defined accumulator are brought into context. The `Accumulator` is constructed with the `"Min"` binary operator. All pre-defined PyGB operators can be constructed manually by the user via similar syntax. `gb.MinPlusSemiring` is the same as `gb.Semiring(gb.MinMonoid, "Plus")`, and `gb.MinMonoid` is the same as `gb.Monoid("Min", "MinIdentity")`. A list of unary and binary operators defined by GBTL is in Fig. 6. These are currently the only operators that can be used in the construction of PyGB operators, but future work is planned to allow more customizable user-defined operators.

```python
def sssp(graph, path):
  with gb.MinPlusSemiring, gb.Accumulator("Min"):
    for i in range(graph.shape[0]):
      path[None] += graph.T @ path
```

(a) PyGB

```
template<class MatrixT, class PathVectorT>
void sssp(MatrixT const &graph, PathVectorT &path) {
  for (GB::IndexType k = 0; k < graph.nrows(); ++k) {
    GB::mxv(path, GB::NoMask(), GB::Min<T>(),
            GB::MinPlusSemiring<T>(),
            GB::Transpose(graph), path);
  }
}
```

(b) GBTL

Fig. 4: Single-source shortest path (SSSP) algorithm

Python's `with` statement provides an abstraction for associating state with a particular block of code. For example, the recommended syntax in Python for file I/O is `with open(...):`, which closes the file automatically at the end of the block. In PyGB, we utilize this language feature to

allow a PyGB operation to infer the operator from its context. An operation will use the corresponding operator with the highest precedence, i.e. lowest nested `with` block with a matching operator. The `Accumulator("Min")` operator can be excluded without changing the behavior of Fig. 4a, since the `path[None] += ...` accumulation step in Fig. 4a will fall back to the `MinMonoid` from the `MinPlusSemiring`. In the case that the accumulation operator is different from the semiring binary operator, the `Accumulator` must be explicitly declared, as in Fig. 7 lines 20-21.

A boolean mask can be applied to the output container during the accumulation step via square bracket notation as in line 4 of Fig. 4a. The default `None` argument (equivalent to `NoMask` in GBTL) will overwrite all data in the pre-existing container with the output from the right-hand-side of the operation. Another PyGB container of the same shape as the output container can replace `None` and its data will be coerced to boolean values, as in Fig. 5a on line 4.

PyGB operators are Python objects that can be constructed as shown in Fig. 6 by providing a string naming a corresponding GBTL function or identity element. Fig. 4a has examples of PyGB operators being constructed ahead of time and then brought into context (`MinSelect1stSemiring`), and being anonymously brought into context (`Accumulator("Min")`). The GraphBLAS C API defines operations on matrices and vectors that are parameterized by these operators. These operations can be invoked from Python using the syntax shown in column 3 of Table I.

Figure 5 shows the triangle counting algorithm implemented in both PyGB and GBTL. This algorithm counts triangles formed by triplets of vertices that are interconnected by performing a matrix-matrix multiply and a reduce to scalar. The Matrix `B` is masked by `L`, the lower-triangular portion of the undirected adjacency matrix, and assigned the output of `L @ L.T`. Reduce uses the `PlusMonoid` from `Arithmetic Semiring` to sum the values of `B`. The corresponding GBTL code is more convoluted, even though the logic is identical.

Fig. 8 shows a more complex algorithm in GraphBLAS, demonstrating nested operator context managers, `apply` and `reduce` operations, constant assignment, and mask complements. In Fig. 7 the `ArithmeticSemiring` is brought into context on lines 20-21, and is subsequently used on lines 22 and 30 for matrix-vector product and vector-vector product operations, respectively. But on line 28 the `BinaryOp("Minus")` operator takes precedence over the semiring. In this example, the operator nesting is primarily for demonstrative purposes, and is questionably beneficial, since it makes no difference to the underlying GBTL and does not improve readability. Nevertheless, this extra nesting is possible and behaves as expected. On lines 25 and 31, the `apply` and `reduce` PyGB operations are used on vectors, and infer the corresponding unary and binary operators the same way as before. On lines 33 and 37, vector assignment is performed using the same square bracket notation used for masks, but providing a Python `slice` object instead of a PyGB container. The expression

```python
def triangle_count(L):
  B = gb.Matrix(shape=L.shape, dtype=L.dtype)
  with gb.ArithmeticSemiring:
    B[L] = L @ L.T
  triangles = gb.reduce(B)
  return triangles
```

(a) PyGB

```cpp
template<class T, class MatrixT>
T triangle_count(MatrixT const &L)
{
  GB::IndexType rows(L.nrows());
  GB::IndexType cols(L.ncols());
  MatrixT B(rows, cols);
  GB::mxm(B, L, GB::NoAccumulate(),
          GB::ArithmeticSemiring<T>(),
          L, GB::transpose(L));
  T triangles = 0;
  GB::reduce(triangles, GB::NoAccumulate(),
             GB::PlusMonoid<T>(), B);
  return triangles;
}
```

(b) GBTL

Fig. 5: Triangle counting algorithm

```python
# unary operators
"Identity"          "AdditiveInverse"
"LogicalNot"        "MultiplicativeInverse"

# binary operators
"LogicalOr"         "LessThan"          "Second"
"LogicalAnd"        "GreaterEqual"      "Min"
"LogicalXor"        "LessEqual"         "Max"
"Equal"             "Times"             "Plus"
"NotEqual"          "Div"               "Minus"
"GreaterThan"       "First"

# example operator constructors
AdditiveInv     = gb.UnaryOp("AdditiveInverse")
PlusOp          = gb.BinaryOp("Plus")
TimesOp         = gb.BinaryOp("Times")
PlusAccumulate  = gb.Accumulator(PlusOp)
PlusMonoid      = gb.Monoid(PlusOp, 0)
ArithmeticSR    = gb.Semiring(PlusMonoid, TimesOp)
```

Fig. 6: using GBTL operators in PyGB

`new_rank[:] = (1.0 - damping_factor) / rows` on line 37 results in the scalar value on the right-hand-side being assigned to all elements of `new_rank`. In line 39, the complement of `page_rank` is aquired by prepending Python's `~` operator to the vector.

An exhaustive list of the operators defined by GBTL is presented in Fig. 6. The DSL can only reference operators defined in GBTL's `algebra.hpp` file. If a user needs to write their own operator while developing an algorithm, they must directly edit the PyGB C++ code. Future work (see Sec. VIII) is planned to support operator definitions from PyGB by reifying Python syntax and translating it to C++.

IV. Design

In this section, we discuss design considerations that influence the PyGB DSL. Our goal is to keep PyGB as close to the GraphBLAS math notation as possible, while working within

```
def page_rank(graph, page_rank,
              damping_factor = 0.85,
              threshold = 1.e-5,
              max_iters = 100000):

  rows, cols = graph.shape
  m = gb.Matrix(shape=graph.shape, dtype=float)
  m[None] = graph
  gb.utilities.normalize_rows(m)
  with gb.UnaryOp("Times", damping_factor):
    m[None] = gb.apply(m)

  page_rank[:] = 1.0 / rows
  new_rank = gb.Vector(shape=page_rank.shape,
                       dtype=m.dtype)
  delta =    Vector(shape=page_rank.shape,
                    dtype=m.dtype)

  while i != max_iters:
    with gb.Accumulator("Second"),
         gb.Semiring(gb.PlusMonoid, "Times"):
      new_rank[None] += page_rank @ m

      with gb.UnaryOp("Plus", (1.0-damping_factor)/rows):
        new_rank[None] = gb.apply(new_rank)

        with gb.BinaryOp("Minus"):
          delta[None] = page_rank + new_rank

        delta[None] = delta * delta
        squared_error = gb.reduce(delta)

    page_rank[:] = new_rank
    if (squared_error / rows) < threshold:
      return page_rank

  new_rank[:] = (1.0 - damping_factor) / rows
  with gb.BinaryOp("Plus"):
    page_rank[~page_rank] = page_rank + new_rank
```

Fig. 7: PageRank in PyGB

the constraints of the Python language. Python's magic methods and dynamic module imports allow for a domain-specific language with cleaner syntax and less boilerplate than C++, increasing ease of use and productivity during the prototyping phase of development. In certain circumstances, we are forced to make pragmatic choices due to limitations of the Python interpreter. Magic methods [18] are a syntax-driven override of certain elements of the Python syntax, such as operators and constructors, and module imports allow processing of file-sized blocks of code. In this section, we focus on the design decisions taken in PyGB, and in Sec. VIII we outline future directions of PyGB design that can provide finer-grained control over PyGB syntax.

The Python DSL exposes nearly all of GBTL's operations through expressions with return values as opposed to the pass-by-reference signatures used in GBTL. Standard matrix algebra is performed via overriding the `+`, `*`, `@` operators. The `assign` and the `extract` operations are performed through the `__setitem__` (`C[i,j] = A`) and the `__getitem__` (`C = A[i,j]`) magic methods, respectively. The `apply` (`C[M] = apply(A)`) and the `reduce` (`c = reduce(A)`) operations are simply function calls. An unfortunate and notable absence from the Python language

```
template<typename MatrixT, typename RealT = double>
void page_rank(MatrixT const     &graph,
               GB::Vector<RealT> &page_rank,
               RealT              damping_factor = 0.85,
               RealT              threshold = 1.e-5,
               unsigned int       max_iters = 100000)
{
  using T = typename MatrixT::ScalarType;

  GB::IndexType rows(graph.nrows()), cols(graph.ncols());
  GB::Matrix<RealT> m(rows, cols);

  GB::apply(m, GB::NoMask(), GB::NoAccumulate(),
            GB::Identity<T,RealT>(), graph);

  GB::normalize_rows(m);

  GB::apply(m, GB::NoMask(), GB::NoAccumulate(),
            GB::BinaryOp_Bind2nd<RealT,
            GB::Times<RealT>>(damping_factor), m);

  GB::BinaryOp_Bind2nd<RealT, GB::Plus<RealT> >
    add_scaled_teleport((1.0 - damping_factor)
                        / static_cast<T>(rows));

  GB::assign(page_rank, GB::NoMask(), GB::NoAccumulate(),
             1.0 / static_cast<RealT>(rows),
             GB::AllIndices());

  GB::Vector<RealT> new_rank(rows), delta(rows);

  for (GB::IndexType i = 0; i < max_iters; ++i)
  {
    GB::vxm(new_rank, GB::NoMask(), GB::Second<RealT>(),
            GB::ArithmeticSemiring<RealT>(),
            page_rank, m);

    GB::apply(new_rank, GB::NoMask(), GB::NoAccumulate(),
              add_scaled_teleport, new_rank);

    RealT squared_error(0);

    GB::eWiseAdd(delta, GB::NoMask(), GB::NoAccumulate(),
                 GB::Minus<RealT>(), page_rank, new_rank);

    GB::eWiseMult(delta, GB::NoMask(), GB::NoAccumulate(),
                  GB::Times<RealT>(), delta, delta);

    GB::reduce(squared_error, GB::NoAccumulate(),
               GB::PlusMonoid<RealT>(), delta);

    page_rank = new_rank;
    if (squared_error/((RealT)rows) < threshold)
    {
      break;
    }
  }

  GB::assign(new_rank, GB::NoMask(), GB::NoAccumulate(),
             (1.0 - damping_factor)/static_cast<T>(rows),
             GB::AllIndices());

  GB::eWiseAdd(page_rank, GB::complement(page_rank),
               GB::NoAccumulate(), GB::Plus<RealT>(),
               page_rank, new_rank);
}
```

Fig. 8: PageRank in GBTL

is an equivalent to the assignment operator of C++. In the Python expression `C = A @ B`, the reference `C` to the pre-existing GBTL container is lost, whereas the equivalent C++ code invokes the `operator=()` method on `C`, preserving the reference to `C`. Although PyGB's default behavior of constructing a new output container is usually correct, it may be an efficient use of resources. Relying on several Python syntactical tricks to reuse containers when possible allows the user to write better-performing code.

Modifying an existing container in-place is desirable when accumulating the result of a sequence of operations or merging multiple containers. To retain a reference to the original container in PyGB, the user can write the expression `C[None] = ...` to invoke the `__setitem__` magic method on `C`. The `None` term corresponds to the GBTL class `NoMask`, and it has no effect on the assigned values, but the invocation of the magic method before assignment ensures that the existing `C` is modified rather than replaced by an entirely new container. The `+=` operator can be used in conjunction with a binary operator to add right-hand-side values to elements already in the left-hand-side container. This will invoke Python's `__iadd__` method on `C`. PyGB users will need a basic understanding of Python and C++ object lifecycles to know when to use `C = A @ B` vs. `C[None] = A @ B`, as the peformance differences between the two are not negligible.

PyGB uses deferred operator evaluation to enable the expression syntax without excessive copying of data. A naive implementation of `C[None] = A + B` might first construct a temporary matrix to store the results of `A + B`, and then assign that temporary matrix into `C`. This will produce the correct output, and calling each C++ operation at the site of the corresponding Python method significantly reduces complexity of the DSL. But the creation of a temporary matrix is costly. To eliminate the copy, the `A + B` operator returns an expression object wrapping the `A` and `B` operands, which then evaluates `A + B` inside of `C.__setitem__()`, without temporaries. The expression object also captures the value of the binary operator from the context of the `A + B` expression (the binary operator is set in an enclosing `with` block) to allow for lazy evaluation. Our lazy expression evaluation paradigm closely corresponds to expression templates in C++ [20], [21] where a compile-time type representation of an expression is generated. In Python, our expression objects are runtime objects that are created through Python syntax reification, using magic methods.

In some circumstances, it is not possible to avoid the intermediate evaluation step due to restrictions of GBTL's evaluation strategy. For example, if a user calculates a matrix-matrix product and assigns it as a submatrix to a larger output matrix, they may write the PyGB code `C[2:4,2:4] = A @ B`. While this PyGB code appears like it should evaluate without a copy, GBTL has no way to express it as a single merged operation. GBTL instead requires two separate operation calls, one to `mxm` and then one to `assign`. This behavior is made opaque to the user by the DSL. There are several other instances throughout the DSL where this extraneous copy is forced. If at some point in the future GBTL enables more sophisticated fusing of operations, it will be trivial to add the support to PyGB. Furthermore, a future GBTL implementation may be able to seamlessly fuse such separate operations through its own lazy evaluation techniques, and such optimization would be immediately available to PyGB users through compiled GBTL code.

Our lazy evaluation system can lead to expressions being left unevaluated indefinitely, so we define a set of terminating operations that force expression evaluation. Any operation that treats the expression like a matrix (assigning to it, extracting from it, combining it with another container) will cause it to be evaluated. The non-blocking mode specified by The GraphBLAS C API has the potential to further improve the performance of lazy evaluation through operation fusion (for example, in the case of element assignment into a sparse matrix). Unfortunately, this non-blocking functionality is not implemented in GBTL, so further exploration of lazy evaluation is left for future work. The deferred expression evaluation system currently employed by PyGB is extensible enough to incorporate future non-blocking optimizations that are added into GBTL.

When the user writes an expression such as `C = A @ B`, the semiring to be used must be declared elsewhere. The tactic of passing the semiring object directly to the `mxm` function, used in GBTL, cannot be used with the expression (operator) syntax we support in PyGB. An alternative method would be to configure all operators at the global scope where every GraphBLAS operation has access to them, but this approach removes functionality present in the C++ operations. Many of the example algorithms use a variety of operators, and mixing and matching them would be more difficult with the approach of a globally determined environment. Even though a semiring may not be used for every operation within a routine, it is often used more than once in close proximity. Specifying the semiring once at the start of a code block where it is used is more concise and easier to understand. Python's context managers provide an abstraction for encapsulating some context within a block. In PyGB, following code brings an operator (monoid in this case) into context:

```
with Semiring(PlusMonoid, "Times"): C = A @ B
```

Behind the scenes, this `with` statement modifies a global stack of operators. Every operation requires an operator of a specific type. When an operation is called, it searches through the stack to find the first operator that it can use. The expression `C = A + B` requires a binary operator. When the expression is evaluated, the `__add__` method finds the `BinaryOp`, `Monoid` or `Semiring` object nearest to its scope. While this approach is intuitive for the types of algorithms we present in this paper, some assumptions must be made about the program execution. There is currently no support for multi-threading with PyGB. In multi-threaded environment, each thread would need to keep track of its own operator stack and other complications would ensue. Due to the Global Interpreter Lock (GIL) in Python, many of the benefits of writing multi-threaded code are already

lost[2]. To circumvent the GIL, many Python programmers use the `multiprocessing` library, which spawns new processes with isolated memory spaces. The use of this library does not interfere with the context managers in PyGB, since the global operator stack is unique to each process. Last, since PyGB compiles to GBTL, it may be more suitable in some situations to use a multithreaded GBTL backend instead of multithreading in Python, but there may be circumstances where a Python program may require sequential GraphBLAS in multiple Python threads.

We discuss some further design issues in Sec. VIII, where we consider near-future PyGB extension options. In the next section, we outline how PyGB DSL is translated to GBTL, compiled, and executed from Python environment.

V. Implementation

The choice of C++ to implement GBTL was a practical one. C++ is "close to the metal," meaning there are fewer levels of abstraction between source code and machine instructions than in higher level languages. C++'s support for templated functions and classes provides a mechanism for generic programming. This means that containers and operations can be written once and reused with different data types. Because template instantiation happens at compile time, there is no runtime performance cost for generic typing. But the dynamic execution model PyGB requires these templated GBTL modules be compiled on the fly. The DSL must inform the compiler which version of the source code should be built at runtime, since Python does not check data types ahead of time.

Python is dynamically typed, which means that variable types are checked at runtime instead of compile time. Python has tools for manually checking the types of objects, such as the built-in `isinstance` and `type` functions, enabling us to implement wrapper functions that pass the Python data type information to the C++ preprocessor. Many high-performance Python libraries allow the programmer to specify a wider range of data types. For example, the NumPy library provides the `dtype` class describing the type, size, and other metadata of a data object. All common C++ Plain Old Data Types have an equivalent NumPy `dtype` class (`bool`, `int8_t`, ..., `int64_t`, `uint_8`, ... `uint_64`, `float`, `double`).

PyGB uses NumPy's `dtype` class to map container types to GBTL backend template types. When two containers of different types are combined in a binary operation, an upcast will be performed automatically according to C++'s upcasting rules, unless the output type is specified by the user. If the data type is not specified during container construction, the DSL will fall back to default Python types: 64-bit `int`s and 64-bit `float`s. The user should explicitly specify the data type at construction time if performance is a concern.

If the number of combinations of data types is limited, precompiling a binary with instantiations for every combination of data types is feasible. But the number of combinations of typed operations grows exponentially with the number of data

[2]https://wiki.python.org/moin/GlobalInterpreterLock

Fig. 9: PyGB execution model

types. The `mxm` operation takes four containers, two inputs, an output, and a mask. Each of these can be any of the 11 plain old data types, resulting in 11^4 combinations. From the 17 built-in binary operators (Fig. 6), there are $17 * 11^3$ accumulator types (specifying two input and one output type), and 1020 semiring types. Each input container can also be transposed, and the complement of the mask can be taken to further increase the number of combinations. There are roughly 6 trillion combinations of template parameters for `mxm` alone, making it infeasible to precompile GBTL templates ahead of time.

While developing the DSL, we discussed possible workarounds for this problem. One solution is to define a C++ union type that stores any of the 11 predefined data types and use GBTL containers of that union type. This approach adds execution overhead and inefficiency, since an additional step is required to look up the container element values. It also prevents us from extending the DSL in the future (see Sec. VIII) to support user-defined types and operators. Another option is to incorporate the Python C API into the C++ binding code, so that container templates are instantiated with the generic `PyObject*` data type. This approach is much more promising for adding support for user-defined data types. Unfortunately the performance will be even worse, since `PyObject` stores auxiliary data for the Python interpreter beyond the primitive value. The solution we settled on was to compile individual GBTL functions just-in-time (JIT) as they are executed in the DSL.

When a binary operation is called from PyGB, the data types of each operand is checked to determine the output type through standard typecasting rules. Then PyGB checks for a compiled C module matching the operation types, first looking in memory and then on disk. If the module has not been compiled, PyGB will instantiate a templated source C++ file with the corresponding data types, compile it with `gcc`, and dynamically load it into the program. The cost of compiling the code can be amortized over future runs of the same code. Furthermore, the program caches the module in memory after it has been imported once.

This compartmentalization of operations into separate modules makes the compiled binaries more reusable, but also causes the program dependencies to grow. A planned feature of the lazy evaluation system would allow a series of operations to be deferred until a single binary module containing all the previously deferred operations is compiled. This improvement will allow a chain of steps in an algorithm to be compiled into a single module. Grouping more operations into a single module will reduce the overhead of function redirection in Python and shorten compile times at the expense of reusability. Having this control over the granularity of module compilation will enable us to experiment in the future with more sophisticated compiler optimizations and extensions to GBTL including template expressions and operation fusion.

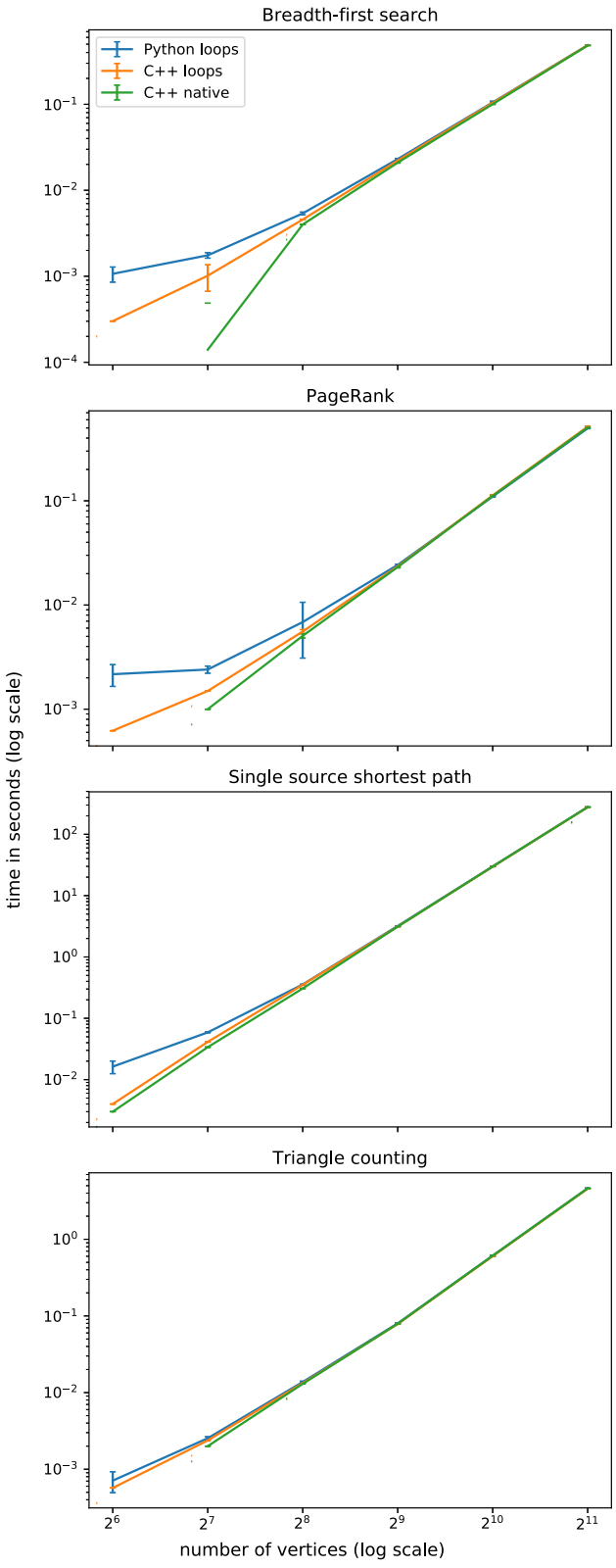

Fig. 10: Performance of four algorithms on Erdős-Rényi graphs with density $|E| = O(|V|^{1.5})$

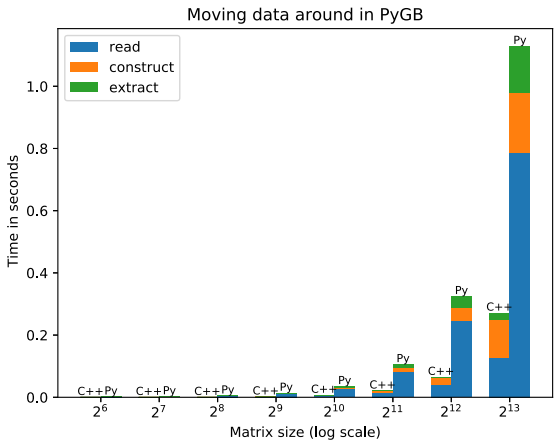

Fig. 11: Operating on `Matrix` objects of various sizes, all with density $|E| = O(|V|^{1.5})$

VI. Experimental Evaluation

Figure 11 shows the time it takes C++ and Python to read a matrix from a file in disk, construct it from a container (`list` in Python and `std::vector` in C++) in memory, and then extract the data back out. The memory management is currently one of the largest penalties PyGB over native GBTL. The file read cost dominates the Python times, but once the matrix has been constructed, operations performed on it from PyGB and GBTL are comparable in performance.

To compare the performance of C++ with Python and measure the abstraction penalty, we implement four algorithms in both GBTL and PyGB and compare the performance with matrices of increasing size. There are three versions of the experiment. In the first version, Python calls C++ operations that were compiled separately, using individual bindings and Python loops. In the second version, Python calls a complete C++ algorithm where the data between GBTL calls is handled by C++. The third version is GBTL C++ native code timed in C++. We implement breadth-first search, PageRank, single-source shortest path, and triangle counting algorithms, and compare the performance for these three versions in Fig. 10.

Breadth-first search has an outer loop that performs `assign` and `mxv` operations, iterating until every vertex has been explored by the frontier vector. PageRank iterates until the rank vector converges to a fixed point, with some specified threshold tolerance. It performs seven GBTL operations in the while loop. Single-source shortest path performs a single `mxv` operation in a loop. Last, triangle counting performs a sequence of operations with no loop. The benchmarks in Fig. 10 indicate a performance penalty associated with the extra Python function redirects, but that penalty becomes less significant as the input scales. This result is expected since the percentage of time spent in GBTL operations increases with input size while the PyGB abstraction penalty remains constant. These experiments show that PyGB is a viable approach to high-performance prototyping with GraphBLAS.

VII. Conclusion

In summary, we have been able to develop a high-level domain-specific language in Python, a popular language for rapid prototyping and scientific computing. The DSL uses an expression syntax that resembles the mathematical notation laid out by the GraphBLAS C API, making it easier to read and write than GBTL. Within the DSL, we use deferred evaluation to enable this syntax, as well as open the door for future work (see Sec. VIII) on the GraphBLAS non-blocking mode.

The implemention of PyGB's interface with GBTL dynamically determines the operand types when an expression is evaluated, and generates templated C++ source code that gets compiled into a binary and executed. This approach allows PyGB to be extensible and support user-defined types and operators in the future (see Sec. VIII). It also allows for code reuse in routines that use the same subset of operations and results in less code to compile when slight modifications are made to PyGB code during development.

The performance of PyGB is comparable to GBTL in the four algorithms that we benchmarked. The overhead of the Python function dispatches is visible at small sizes, but becomes negligible as the matrix size increases. We therefore believe that the DSL is an effective alternative to GBTL that achieves the goals of implementing the functionality of GraphBLAS and simplifying the development of GraphBLAS routines.

VIII. Future Work

Currently, the biggest hindrance to using PyGB is the complexity of the build system and GBTL dependencies required by the Python library. The library depends on `gcc` and `cmake`. Making PyGB portable and easily installable through Python's package manager `pip` will require us to revisit our compiler toolchain.

Several major features of GBTL are still missing from PyGB. The ability for users to define custom data types for the `Matrix` and `Vector` classes paves the road for future implementions of complex numbers and even more exotic types. One interesting prospect is to define sets as the data type of a matrix, and a semiring that performs set unions and intersections. Another missing feature is user-defined operators for use in the PyGB operations. Implementing this feature requires either using an intermediate language such as Cython or forcing the user to write code directly in C++.

Performance of PyGB is a top priority. There are a few ways to address the disparity between Python and C++ when it comes to moving data around in memory. Since a common scenario when using the PyGB will be getting data into and out of the `Matrix` and `Vector` objects, it is desirable to make that operation faster. Figure 9 shows that the majority of the time getting data into a PyGB container comes from reading from disk. C++ is much faster at this operation, and wrapping a C++ function to directly load a matrix instead of first loading into Python lists would be trivial.

There are also circumstances where a matrix already lives in memory in a different container. Python has an array buffer API for sharing contiguous blocks of memory that would enable

PyGB to construct a container without copying all the data. This complicates resource ownership and garbage collection between Python and C++, but is something worth exploring.

ACKNOWLEDGMENTS

Scott McMillan was 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 [DM18-0390]. The authors affiliated with DOE Pacific Northwest National Laboratory (PNNL) were supported in part by the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit Program and the High Performance Data Analytics Program (HPDA) at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. The authors affiliated with the University of Washington were in part supported by the NSF grant 1642439.

REFERENCES

[1] "The Graph BLAS Forum," 2018. [Online]. Available: http://graphblas.org/

[2] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for Graph Algorithm Primitives," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2013, pp. 1–2.

[3] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical Foundations of the GraphBLAS," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2016, pp. 1–9, 00018.

[4] J. Kepner, "GraphBLAS Mathematics - Provisional Release 1.0 -," GraphBLAS.org, Tech. Rep., Apr. 2017. [Online]. Available: http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf

[5] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The GraphBLAS C API Specification," GraphBLAS.org, Tech. Rep., 2017. [Online]. Available: https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C.pdf

[6] Jeremy Kepner and John Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*, ser. Software, Environments and Tools. Society for Industrial and Applied Mathematics, Jan. 2011.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990.

[8] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[9] T. Davis, "SuiteSparse : A Suite of Sparse Matrix Software," http://faculty.cse.tamu.edu/davis/suitesparse.html, 2018.

[10] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear Algebra Graph Kernels for NoSQL Databases," in *Proc. IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 822–830.

[11] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (D4M) database and computation system," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2012, pp. 5349–5352.

[12] K. Ekanadham, B. Horn, J. Jann, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph Programming Interface: Rationale and Specification," IBM Research Division, Thomas J.Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 USA, Tech. Rep. RC25508 (WAT1411-052), Nov. 2014.

[13] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "GraphPad: Optimized Graph Primitives for Parallel and Distributed Platforms," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 313–322.

[14] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "GBTL-CUDA: Graph Algorithms and Primitives for GPUs," in *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, pp. 912–920. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7529957

[15] "GraphBLAS Template Library (GBTL)," https://github.com/cmu-sei/gbtl, 2018.

[16] "Python," https://www.python.org/, 2018.

[17] "Jupyter," http://jupyter.org/, 2018.

[18] B. Klein, "Magic methods and operator overloading," https://www.python-course.eu/python3_magic_methods.php, 2018.

[19] "`contextlib` — Utilities for `with` -statement contexts," https://docs.python.org/2/library/contextlib.html, 2018.

[20] T. Veldhuizen, "Expression Templates," *C++ Report*, vol. 7, no. 5, pp. 31, 26, 1995.

[21] D. Vandevoorde and N. Josuttis, *C++ Templates – the Complete Guide*. Addison-Wesley, 2003.