GBTL-CUDA: Graph Algorithms and Primitives for GPUs

Peter Zhang, Marcin Zalewski, Andrew Lumsdaine
Center for Research in Extreme Scale Technologies (CREST)
Indiana University
Bloomington, IN
Email: {pazhang,zalewski,lums}@indiana.edu

Samantha Misurda, Scott McMillan
Software Engineering Institute (SEI)
Carnegie Mellon University
Pittsburgh, PA
Email:{slmisurda,smcmillan}@sei.cmu.edu

***Abstract*—GraphBLAS is an emerging paradigm for graph computation that makes it easy to program new graph algorithms in a highly abstract *language of linear algebra*. The promise of GraphBLAS is that an abstract graph program will execute in a wide variety of programming environments, ranging from embedded environments to distributed memory computers. In this paper we present our initial implementation of GraphBLAS primitives for graphics processing unit (GPU) systems called GraphBLAS Template Library (GBTL). Our implementation is an ongoing effort in the context of GraphBLAS standardization efforts by a diverse group of academics and representatives of the industry. Our implementation consists of a high-level C++ frontend, and the GPU functionality is implemented with a combination of the CUSP library for sparse-matrix computation on GPU and the NVIDIA Thrust framework for abstract GPU programs. We give initial performance results of our implementations, and we discuss solutions to the problems we encountered when providing a low-level implementation for a high-level generic interface.**

I. Introduction

Graph algorithms are essential to many modern-day computer applications. The growing complexity of the problems to be solved by graph algorithms creates a new set of challenges. Given the increasing scale of recent problems such as social networks, cybersecurity, health informatics, gene sequence assembly, and artificial intelligence, heterogeneous accelerators are increasingly being used to speed up graph algorithm computation [5, 7, 9, 14]. One commonly used accelerator is the GPU, which delivers high throughput through high memory bandwidth and a massively-parallel, single instruction, multiple data (SIMD) execution model. Implementing efficient graph algorithms on the GPU to take advantage of these features, however, often requires a deep knowledge of GPU architecture and programming.

To date, most graph algorithms that are implemented on the GPU are specialized, such as [17] and [4] in the case of breadth-first search (BFS). Although they often deliver relatively good performance, the fact that these algorithms are specialized and hand-optimized makes them difficult to reuse and apply to other applications. There also exist libraries that abstract operations commonly used on the GPU and optimize these operations in order to achieve good performance, such as *Gunrock* [16] and *back40computing* [13]. Of these, the latter delivers the best performance on BFS algorithms on the GPU. Both libraries, however, extract a fairly high-level abstraction of the operations specifically optimized for execution on the GPU, which might not be applicable to other accelerators such as the Intel® Xeon Phi™. In addition, the set of operations provided by the libraries that compose a graph algorithm might not be the most intuitive combination to someone with no prior knowledge of GPU programming and of the libraries themselves.

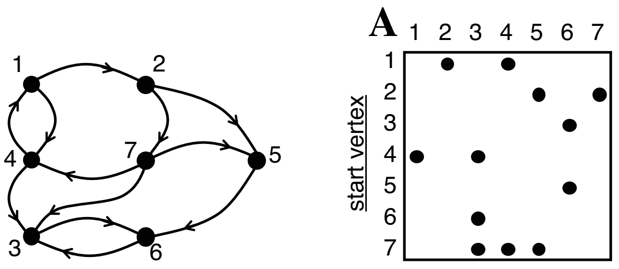


Fig. 1: Sparse adjacency matrix representation of a graph [10].

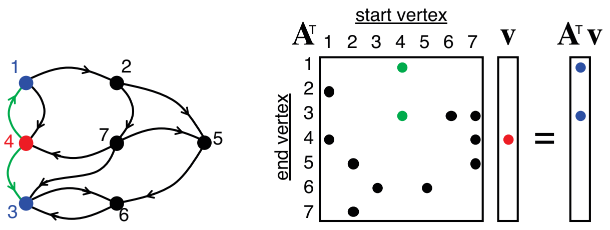


Fig. 2: One iteration of BFS search; source is vertex 4 [10].

The GraphBLAS Forum is currently working to solve this problem by specifying a set of primitives (data structures and operations) that will allow graph algorithms to be described using linear algebra, which is natural when the graphs are represented by means of an adjacency matrix (Figure 1)[1, 12]. For example, the BFS algorithm can be represented simply by a matrix-vector multiplication between the graph and a frontier vector that represents the vertices being visited at a particular level (Figure 2). The multiplication can be optimized differently depending on the hardware on which it executes, but the operation signature remains the same across different platforms.

Operation	Description
buildMatrix	Build a sparse matrix from row, column, value tuples
extractTuples	Extract the row, column, value tuples from a sparse matrix
mXm, mXv, vXm	Perform sparse matrix multiplication (e.g., BFS traversal)
extract	Extract a sub-matrix from a larger matrix (e.g., sub-graph selection)
assign	Assign to a sub-matrix of a larger matrix (e.g., sub-graph assignment)
eWiseAdd, eWiseMult	Element-wise addition and multiplication of sparse matrices (e.g., graph union, intersection)
apply	Apply unary function to each non-zero element of matrix (e.g., edge weight modification)
reduce	Reduce along columns or rows of matrices (vertex degree)
transpose	Swaps the rows and columns of a sparse matrix (e.g., reverse directed edges)

TABLE I: Current list of core GraphBLAS operations [10].

This effort is preceded by or coincides with a number of attempts at implementing linear algebraic primitives similar to the GraphBLAS proposal. Notable is the Combinatorial BLAS (CombBLAS) that implements operations on a distributed CPU system using Message Passing Interface (MPI) [3]. In another approach, Graphulo implements GraphBLAS building blocks on top of the NoSQL library Apache Accumulo [8]. The Graph Programming Interface (GPI) is similar to GraphBLAS, whose goal is to achieve portability across a wide variety of architectures [6].

As a part of this effort, we have designed and are implementing a C++ library with a modular structure that enables GraphBLAS operations to also be implemented on different architectures, and in this paper we focus on the effort to take advantage of the GPU's computing power and memory bandwidth. More specifically, we make the following contributions:

- We develop a C++ template library that implements the current iteration of the GraphBLAS API (operations and data structures) called GBTL that represents the common interface to multiple, selectable backends. Having multiple, selectable backends allows for device-specific optimizations within these backends for different hardware architectures.
- We implement two backends: a simple sequential CPU implementation as well as an NVIDIA CUDA®-based backend built on top of the Thrust and CUSP libraries for GPU. We present the latter in this paper.
- We implement a number of graph algorithms, such as BFS and single source shortest paths (SSSP), to demonstrate the applicability and simplicity of these algorithms using GraphBLAS. We show results for the CUDA backend.

Our code, which is under active development, can be found on GitHub at https://github.com/cmu-sei/gbtl.

II. Background

A. GraphBLAS

Just as the Basic Linear Algebra Subprograms (BLAS) interface of dense linear algebra has been used for decades as the API for simulation and modeling codes, the purpose of the GraphBLAS initiative is to define a set of primitives (data structures and operations) and hence, the API, for graph algorithms and applications. The primary difference is that GraphBLAS is based on semiring algebra using sparse matrices. As in BLAS, the GraphBLAS API is able to separate the concerns between graph algorithms and application development and the increasing complexity of developing highly-tuned implementations on heterogeneous architectures.

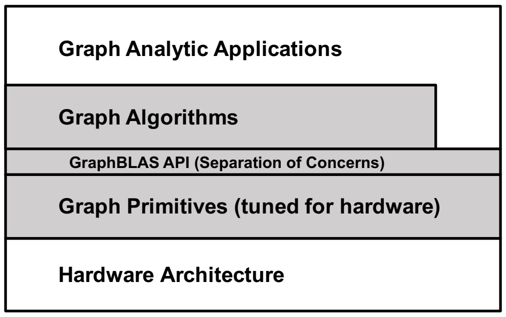

Graph Analytic Applications
Graph Algorithms
GraphBLAS API (Separation of Concerns)
Graph Primitives (tuned for hardware)
Hardware Architecture

Fig. 3: The GBTL software architecture (shaded in gray) implements the GraphBLAS API tuned for specific hardware architecture below the separation of concerns, and includes algorithms using the GraphBLAS API above.

The current set of core GraphBLAS operations in this API is defined in Table I, and the sparse matrix is the key data structure used for storing the graph. Unlike dense matrices, sparse matrices have unstored elements that are referred to as *structural zeros*, and these elements are not accessed during the operations on these matrices. In addition, operations such as matrix multiplication are not restricted to the standard arithmetic multiply and addition operations as those in dense linear algebra, allowing for different operations to be substituted; these different operations often, but do not always, conform to the properties of semiring algebra [11]. For example, the standard arithmetic semiring is similar to that used in standard linear algebra and uses the multiply and addition operations where "zero" is defined as the additive identity and multiplicative annihilator, and "one" is defined as the multiplicative identity. We use the notation $\langle D, +, \times, 0, 1\rangle$ to define this semiring for scalars in domain D. Another popular semiring in GraphBLAS is where multiplication and addition are replaced with addition and minimum respectively, given with the notation $\langle D, min, +, \infty, 0\rangle$.

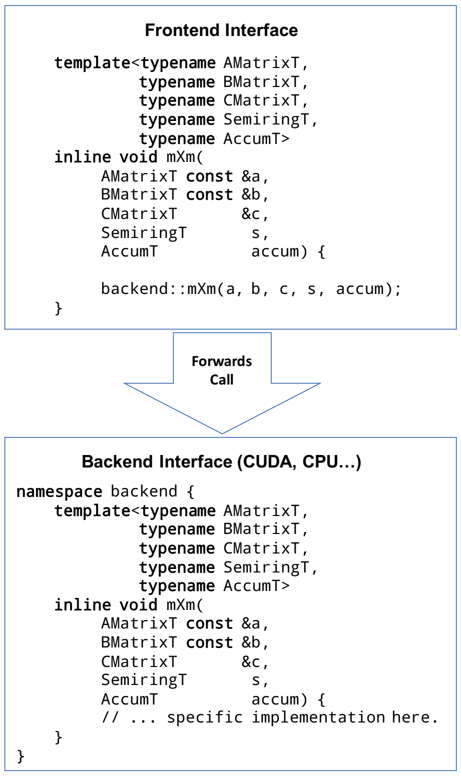
B. Thrust and CUSP Libraries

The Thrust library is a software package that is shipped with NVIDIA's CUDA toolkit. It has an interface similar to the C++ STL including STL-like data structures such as a `device_vector`, which has similar interfaces as `std::vector` but resides on the GPU.

CUSP is an open source library built on top of Thrust that provides support for sparse matrices and related operations. We use the CUSP library to implement our CUDA backend and use the coordinate list (COO) matrix format provided by CUSP as our default sparse matrix type. The COO matrix format contains three vectors: `row_indices`, `column_indices` and `values`; each corresponding entry in those vectors indicates one stored value in the matrix. This format allows for fast conversion to and from other formats such as compressed sparse column (CSC) or compressed sparse row (CSR).

III. GBTL: GraphBLAS Template Library

A. Library Design



```
    template<typename AMatrixT,
             typename BMatrixT,
             typename CMatrixT,
             typename SemiringT,
             typename AccumT>
    inline void mXm(
         AMatrixT const &a,
         BMatrixT const &b,
         CMatrixT       &c,
         SemiringT       s,
         AccumT          accum) {

         backend::mXm(a, b, c, s, accum);
    }
```

```
namespace backend {
    template<typename AMatrixT,
             typename BMatrixT,
             typename CMatrixT,
             typename SemiringT,
             typename AccumT>
    inline void mXm(
         AMatrixT const &a,
         BMatrixT const &b,
         CMatrixT       &c,
         SemiringT       s,
         AccumT          accum) {
         // ... specific implementation here.
    }
}
```

Fig. 4: GBTL Separation: templates and forwarding of GraphBLAS operations are used to connect the frontend to the backend.

The design of GBTL (Figure 3) captures the goal of separating concerns between algorithm design and hardware-specific optimization by establishing the GraphBLAS API as the boundary between the algorithms that use this API (in the *frontend*) and hardware-specific implementations of the API (in the *backend*). In this approach, the frontend algorithms do not have hardware-specific code, and the backend is selected at compile time with an appropriate compiler flag. Different backends provide different strength and flexibility geared towards a diverse combination of applications and environments. Therefore, graph algorithms can be implemented in terms of the GraphBLAS API with no need for hardware tuning.

The API implements all of the core GraphBLAS operations listed in Table I. After being invoked, the frontend function forwards the call to the selected backend, which has the same interface as all the other backends (Figure 4). The backend interface then executes a backend-specific implementation of the GraphBLAS operation. To ensure interoperability, the expected behavior of any specific operation is preserved across different backends. We currently have more than one backend implementation to ensure that separation of concerns is achieved: a naïve single CPU implementation and a GPU implementation that we discuss in more detail in this paper.

The Matrix class used in the GraphBLAS operations is an opaque data structure in the frontend, as shown in the frontend declaration of the Matrix template in Figure 5. The use of variadic templates in object instantiation enables the user to provide hints about the properties of the matrix they are constructing, thus allowing the backend to optimize specifically for best performance. For example, if a matrix object is constructed using the following template parameters:

```
    Matrix <double, DenseMatrixTag,
        DirectedMatrixTag> matrix(...);
```

the backend can choose the matrix data structure that provides the best support for a dense, directed matrix. It is up to the specific implementation of the backend to utilize different dense or sparse matrix formats, such as CSR or COO, that could even be implemented as user-defined types in third-party packages. Figure 4 also illustrates how we use template parameters to avoid the need to expose these backend types to the API definitions or the algorithms being implemented in the frontend. Although we allow the user to provide hints during construction of a sparse matrix, the backend implementation selects the specific format. The final data type chosen by the backend is influenced by hardware-specific memory structures and other features, and the desire to achieve optimal performance.

B. GBTL-CUDA Backend

In this paper, we will focus on the GPU-based CUDA backend, which is currently implemented on top of two third-party libraries: CUSP and Thrust. The Thrust library provides GPU programming primitives that enable the implementation of GraphBLAS operations without having to tune individual operations by hand. Built on top of Thrust, the CUSP library consists of parallel algorithms and data structures for sparse linear algebra and is implemented on top of Thrust as well. The hierarchical relationship between the GBTL-CUDA backend interface, CUSP, and Thrust can be found in Figure 6. This section serves as a brief description of how GraphBLAS operations are implemented using these two libraries.

1) negate (structural complement): Although not specified in the GraphBLAS specifications, we implement a *negate* function that takes a matrix as an input argument and returns the structural complement where stored values are replaced with structural zeros and structural zeros are replaced with the

```
// TagsT template parameters provide hints
template <typename ScalarT, typename... TagsT>
class Matrix :
    public backend::Matrix<ScalarT, TagsT...>
{
public:
    typedef ScalarT ScalarType;

    // Empty construction; fixed dimensions
    Matrix(IndexType num_rows, IndexType num_cols);

    // Other frontend matrix interface...

private:
    // immutable dimensions:
    IndexType const m_num_rows, m_num_cols;

    // opaque backend implementation
    backend::Matrix<
            ScalarType,
            detail::SparsenessCategoryTagT,
            TagsT...>
        m_matrix;
};
```

Fig. 5: The frontend Matrix template in GBTL.

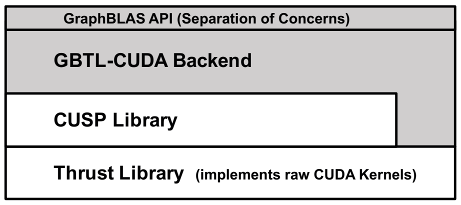

GraphBLAS API (Separation of Concerns)
GBTL-CUDA Backend
CUSP Library
Thrust Library (implements raw CUDA Kernels)

Fig. 6: CUDA backend software architecture (the Graph Primitives block from Figure 3 tuned for GPU).

multiplicative identity. For example, the result of negation of the following matrix, where dash (−) indicates a structural zero:

$$A = \begin{bmatrix} 2 & - & - \\ - & 1 & 3 \end{bmatrix}$$

becomes

$$negate(A) = \begin{bmatrix} - & 1 & 1 \\ 1 & - & - \end{bmatrix}$$

relative to the arithmetic semiring, $\langle D, +, \times, 0, 1\rangle$, where the multiplicative identity, 1, is used for the complement. We find the negation operation to be useful in both BFS and SSSP algorithms.

2) buildMatrix and extractTuples: The operations *buildMatrix* and *extractTuples* are the only two methods specified in GraphBLAS that allow for direct access to the stored values of the matrices. In the CUDA backend, these are implemented using `thrust::copy` calls that allow for host-to-device and device-to-host memory copies of these values and their corresponding indices from the COO matrix.

3) apply: Since the *apply* operation is essentially an element-wise operation on all stored values in the matrix, a simple "transform" operation on all the stored values of the matrix is sufficient, as represented in Figure 7.

```
apply(matrix, unary_func){
    thrust::transform(matrix.values, unary_func);
}
```

Fig. 7: Pseudocode for apply.

4) eWiseAdd and eWiseMult: These functions perform element-wise binary operations on two source matrices with the same dimensions. Despite their similarity in name, element-wise addition and element-wise multiplication are implemented differently in order to correctly handle the sparse structure of the matrices being operated on.

```
template<typename AMatrixT,
         typename BMatrixT,
         typename CMatrixT,
         typename MonoidT =
             graphblas::math::Plus<typename
                 AMatrixT::ScalarType>,
         typename AccumT =
             graphblas::math::Assign<typename
                 CMatrixT::ScalarType> >
inline void eWiseAdd(
                AMatrixT const &a,
                BMatrixT const &b,
                CMatrixT       &c,
                MonoidT         monoid = MonoidT(),
                AccumT          accum = AccumT());
```

Fig. 8: Interface for eWiseAdd.

Element-wise addition is implemented through merging the row, column, and value vectors of the two source matrices. At locations where both source matrices have stored values, two values appear in the merged list; these values are reduced using the "add" function specified in the GraphBLAS operation's interface, which defaults to arithmetic addition (plus) but can also be user-defined (`monoid` argument in Figure 8). Values stored at a given location in only one of the source matrices only appear in the merged list once, and therefore, do not need to be reduced.

In the actual implementation we perform a symmetric set intersection because the one-way `intersection_by_key` from Thrust only returns values from one of the sets. For the reduction by the same index pairs to work, we need the values from both sets without having to do searches.

On the other hand, element-wise multiplication requires the values that have row and column indices unique to only one matrix to be purged. To do this, we first find the intersection of the index pairs from both matrices to give us the values with coinciding row-column combinations in both matrices, I_{common}. Or in set notation, let $I_A = \{(i_A[n], j_A[n]) | n \in [0, len(i_A))\}$ where i_A and j_A are respectively the vectors of row and column indices from the COO representation of the matrix A (and I_B is similarly defined for matrix B). Then, I_{common} is defined as follows:

```
template<typename AMatrixT,
         typename BMatrixT,
         typename CMatrixT,
         typename MonoidT =
             graphblas::math::Times<typename
                 AMatrixT::ScalarType>,
         typename AccumT =
             graphblas::math::Assign<typename
                 CMatrixT::ScalarType> >
inline void eWiseMult(
                AMatrixT const &a,
                BMatrixT const &b,
                CMatrixT       &c,
                MonoidT        monoid = MonoidT(),
                AccumT         accum = AccumT());
```

Fig. 9: Interface for eWiseMult.

$$I_{common} = I_A \cap I_B.$$

However, Thrust only provides a one-way `intersection_by_key` operation that returns values from one of the sets. For the reduction of values at the same indices to work, we need to the values from both sets. Application of the `intersection_by_key` ($\cap_{key}$) twice accomplishes the necesary computation without using searches as follows:

$$I_{common} = (I_A \cap_{key} I_B) \cup (I_B \cap_{key} I_A).$$

We then reduce all the value pairs from the source matrices that have the same row-column combinations as indicated in I_{common} through the "multiply" function in the GraphBLAS eWiseMult operation's interface (`monoid` argument in Figure 9). As a result, the values at any given location are reduced and stored into the destination if and only if both values from the source matrices are not structural zeros.

5) assign: The *assign* operation assigns values from a source matrix into locations specified by subsets of row and column indices in the destination matrix. Its implementation can be represented by the pseudocode in Figure 10.

```
assign(src, iterator_row, iterator_col, dst, accum){
    //pick out intersection of
    //(iterator_row, iterator_col) with src indices

    vector selected_rows <--
        iterator_row[src.row_indices]
    vector selected_cols <--
        iterator_col[src.col_indices]

    dst.values <-- eWiseAdd(
                    dst.values,
                    thrust::sort_by_key(
                        tuple(selected_rows,
                              selected_cols),
                        src.values),
                    accum)

    dst.row_indices <-- eWiseAdd(dst.row_indices,
                              selected_rows,
                              accum)
    dst.col_indices <-- eWiseAdd(dst.col_indices,
                              selected_cols,
                              accum)
}
```

Fig. 10: Pseudocode for assign.

Since the specification for *assign* does not currently dictate the exact behavior when some of the selected values in the source are structural zeros, there can be two possible outcomes of the assignment to the destination:

- If there are structural zero values selected in the source to be assigned to locations in the destination where the values are not structural zeros, those values in the destination become structural zeros after this operation and are thus annihilated.
- Instead of being annihilated, a value in the destination is retained when a structural zero is assigned to it from the source, and thus the structural zero is the additive identity in the assignment.

In order to implement the behavior described in the first item on a sparse matrix data structure, all the structural zeros that must be assigned into the destination matrix must have their corresponding values erased in order to effectively assign sparsity, which requires remove and/or copy functions and impacts performance of the operation. We currently implement the second behavior listed above.

6) extract: The *extract* operation allows for selection of a sub-graph from a source matrix based on a combination of subsets of row and column indices specified by a row index vector (`i`) with p elements and a column index vector (`j`) with r elements. We implemented the extract operation using an algorithm specified in [2, 10]:

- First, construct a $p \times m$ selection matrix, S_i, where each row has only one element equal to the multiplicative identity (usually 1), whose column is $i[row]$.
- Second, construct an $n \times r$ selection matrix, S_j^T, where each column has only one element equal to the multiplicative identity, whose row is $j[column]$.
- Finally, the $p \times r$ result of the extraction from an $m \times n$ matrix A with the given row and column indices i and j can be found through the following matrix multiply operations:

$$result = (S_i \times A) \times S_j^T$$

7) reduce (row, column, matrix): Reduction can be easily achieved through the `thrust::reduce` class of functions. The differences between row, column, and whole-matrix reductions are that both row and column reductions result in a vector, while matrix-level reduction results in a single value (the scalar type of the matrix). For example, reduction by row can be implemented as the pseudocode in Figure 11.

```
row_reduce(src, dst, binary_func){
    dst <-- thrust::reduce_by_key(
            key = src.row_indices,
            values = src.values,
            binary_func)
}
```

Fig. 11: Pseudocode for reduce.

8) vXm, mXv, mXm and transpose: Since *vXm* and *mXv* operations are operated the same way as *mXm* operations where one of the arguments has only one row or column, all three operations are implemented in the same manner. We simply forward the matrix multiplication and transpose operations onto the same functionalities provided by CUSP. Note that CUSP allows the traditional multiplication and addition operations to be overridden, and in this way semiring operations are supported as well as arbitrary user-defined function pairs. GPU-specific device functions may need to be implemented to support accelerated versions of the overridden operations. For example, in a BFS application where the user might need to override the multiplication operation with a `select_first` function and the addition with a `min` function, which corresponds to the semiring $\langle D, min, select\_first, \infty, 0\rangle$, a binary CUDA device functions will need to be implemented for `select_first`, as shown in Figure 12.

```

struct select_first{
    template <typename T>
    __host__ __device__
    T & operator()(T &lhs, T &rhs){
        return lhs;
    }
};
```

Fig. 12: CUDA device function for `select_first`.

IV. Discussion

We test our implementation of the GBTL library's CUDA backend using BFS and SSSP algorithms on a number of graphs from the University of Florida's Sparse Matrix Collection [15], and we perform individual benchmarking of a few selected operations. We perform our experiments on a dual-socket Intel Xeon E5-2670 v3 12-core processor machine with 32 GB RAM and an NVIDIA Tesla K40M GPU.

A. mXm Scaling

We perform scaling tests on the *mXm* operation, which is used in all of the algorithms we have seen so far. We use randomly generated Erdős-Rényi graphs with the number of non-zero values set to $0.05 \times n^2$ where n is the number of vertices, so that both arguments of the product have the same sparsity. We then evaluate performance by comparing the number of millions of arithmetic (floating point) operations per second of runtime (MFLOPS) across matrices of different scales. The performance for *mXm*, as displayed in Figure 13, scales fairly well up until dimension 1638×1638 and plateaus until 4096×4096, at which point we run out of GPU memory.

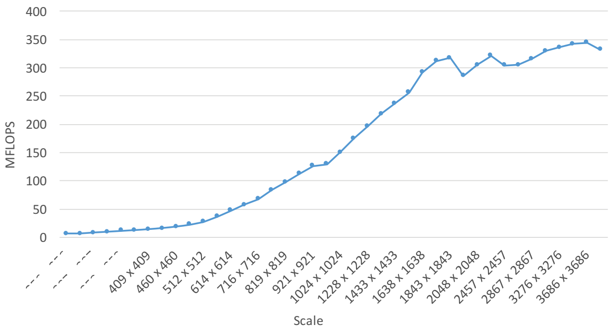


Fig. 13: mXm Scaling in MFLOPS.

B. Example GraphBLAS Algorithms

```
// wavefront initialized with root vertex = 1
bfs(graph, wavefront, level) {
    int previous_nnz = 0;
    graph = transpose(graph);
    while (wavefront.nnz() > previous_nnz) {
        previous_nnz = wavefront.nnz();
        wavefront = mXv(graph,
                        wavefront,
                        AlgebraicSemiring);
    }
}
```

Fig. 14: Simple BFS.

```
// wavefront initialized with root vertex = 0
bfs(graph, wavefront, level) {
    vector visited = wavefront

    while(!wavefront.empty()) {
        // increment level in next wavefront
        wavefront = vXm(wavefront,
                        graph,
                        Add1NotZero)

        // filter out already visited vertices
        // if the vertices have values less than
        // current level in visted vector
        wavefront = eWiseMult(wavefront,
                              negate(visited),
                              Mult)

        //update visited vector by filtered wavefront
        visited = eWiseAdd(wavefront,
                           visited,
                           throwException)
    }
}
```

Fig. 15: GraphBLAS level-BFS implementation.

Breadth-First search (BFS) is one of the simplest algorithms to implement in GraphBLAS. Kepner and Gilbert [11] define

BFS simply as a repeated multiplication of a transpose of an adjacency matrix $\mathbf{A}$ with a wavefront column vector, $\mathbf{v}$ (Figure 2). The rows of the transpose $\mathbf{A}^T$ represent incoming edges, where the index of the row is the target vertex and the index of the column is the source vertex. Multiplication of a given row with the vector yields a non-zero value if any of the incoming edge sources matches a vertex marked in the wavefront $\mathbf{v}$. Each multiplication generates a new wavefront, and BFS continues until the wavefront stops growing (i.e., stops reaching new vertices). This simple version of the algorithm translates to the GraphBLAS pseudocode shown in Figure 14.

Such a simple BFS, however, can only indicate which vertices are reachable from the source vertex. To obtain some useful information from BFS, we can, for example, compute the level in which a given vertex was reached as shown in Figure 15. The vector-matrix operation, `vXm`, on line 7 in Figure 15 takes the wavefront row vector as the left-hand operand (the values `wavefront` are the levels) and multiplies it with the matrix representation of the graph. By using `vXm` instead of `mXv` as shown in Figure 14, we save one transpose operation.

The semiring argument for this `vXm` operation has been set to `Add1NotZero`, and the "multiply" and "add" operations that are contained in it are shown in Figure 16. For its multiplication operation, shown on line 1, the first (left-hand) argument is incremented if it is not equal to 0. The check of zero is used to handle the case when the wavefront vector is dense. When the vector is sparse, the check is not needed. The addition operation of the semiring returns the first operand, as shown on line 6. In principle, this addition operation can return either value because level values stored in the wavefront vector are always the same for every vertex at the same level.

```
1 Mult(first, second)
2 {
3     return first == 0 ? 0 : ++first;
4 }
5
6 Add(first, second)
7 {
8     return first;
9 }
```

Fig. 16: Pseudocode for the multiplication and addition operations of the `Add1NotZero` "semiring" for the algorithm shown in Figure 15.

On line 14 (Figure 15) we filter out the vertices that were already visited from the `wavefront`, which are indicated by stored values in the `visited` vector. The `negate` modifier finds the structural complement of the `visited` vector for the purpose of this `eWiseMult` operation (the negation is not materialized). Every value that did not exist (was sparse) in `visited` is turned into the multiplicative identity of the `Mult` (multiplication) operation, and every value that exists (not sparse) is sparse in the negation. This way, `eWiseMult` functions as a set difference between the elements of the `wavefront` and the elements of `visited` (wavefront \ visited). As discussed in the next subsection, we actually implement this flavor of `eWiseMult` with `thrust::set_difference`, a GPU-implemented set difference operation from Thrust.

Graph Name	# Vertices	# Edges	Runtime(ms)	MTEPS(*)
Journals	124	12,068	5.76	2.1
G43	1,000	19,980	14.61	1.4
ship_003	121,728	3,777,036	558.95	6.8
belgium_osm	1,441,295	3,099,940	10502.4	0.3
roadNet-CA	1,971,281	5,533,214	4726.21	1.2
delaunay_n24	16,777,216	100,663,202	65507.7	1.5

TABLE II: BFS Runtime on Real-World Graphs.
*MTEPS = **M**illions of **T**raversed **E**dges **P**er **S**econd

Finally, on line 19, we update `visited` with the values from the `wavefront`. Note that `wavefront` is already filtered by `visited`, so there will always be only one value to take, either from `visited` or from `wavefront`. Accordingly, we use the `throwException` operation for the `eWiseAdd`'s `monoid` argument, which throws an exception if it is actually invoked. Note that `eWiseAdd` can be specialized for this operation, and in non-debug builds of GraphBLAS programs it can be optimized down to a set union between the two vectors.

The example of BFS GraphBLAS code shows how the linear algebra notation can be used to implement algorithms that perform interesting side effects. The operations that we use in this implementation are not true mathematical semirings and monoids, but they match the relaxed expectations that GraphBLAS places on its operands and thus can be used to implement the necessary functionality. In general, to implement interesting algorithms, one must often come up with mathematically unusual entities.

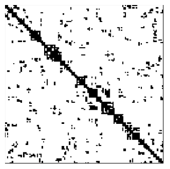
C. Breadth-First Search Implementation

Our BFS implementation is specified in Section IV-B. The only difference is that instead of using a direct *eWiseMult* operation on the filtering process, which is the second GraphBLAS call in the pseudocode given on line 14 in Figure 15, we used the specialized Thrust method, `thrust::set_difference`, for filtering, as discussed in the previous subsection.

Some interesting trends can be found in the result. As expected, the performance of BFS depends on the attributes of the graph itself. Even though both are road network graphs, the performance of BFS on *belgium_osm* seems to lag behind *roadNet-CA* despite having fewer vertices and edges. The sparsity plot of the two graphs, where the black dots are a magnified representation of zeros in the matrix, are shown in Figure 17. We can observe that the sparsity pattern of the *roadNet-CA* graph is quite different from *belgium_osm*. The *belgium_osm* graph has fewer edges per vertex connected to other vertices than *roadNet-CA*, which results in a greater runtime for BFS to reach all vertices.

D. Single-Source Shortest Paths Implementation

We are able to further test the functionalities of the library using a simple, naïve SSSP algorithm, which requires the construction of an identity matrix. The complete code for the SSSP implementation can be found in Figure 18. Even simpler


(a) Sparsity of *roadNet-CA*

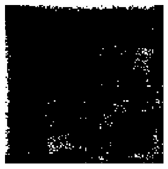

(b) Sparsity of *belgium_osm*

Fig. 17: Comparison of sparsity between *roadNet-CA* and *belgium_osm*. Black dots represent zeros in matrix[15].

than BFS, the algorithm can be implemented with just one *mXm* operation using a specialized accumulation function.

```
void sssp(MatrixT const      &graph,
          PathMatrixT const &start,
          PathMatrixT       &paths)
{
    using T = typename MatrixT::ScalarType;
    using MinAccum =
        graphblas::math::Accum<
            T,
            graphblas::math::ArithmeticMin<T> >;

    paths = start;

    graphblas::IndexType rows, cols;
    graph.get_shape(rows, cols);

    for (graphblas::IndexType k = 0; k < rows; ++k)
    {
        graphblas::mXm<
            PathMatrixT,
            MatrixT,
            PathMatrixT,
            graphblas::MinPlusSemiring<T>,
            MinAccum>(paths, graph, paths);
    }
}
```

Fig. 18: Code for SSSP.

V. Future Work

Since the GraphBLAS Forum is an ongoing standardization effort and the specifications for the operations are still currently in draft, there are a lot of possibilities for the future evolution of the GBTL project. We have demonstrated a working GraphBLAS implementation on the GPU, as well as the applicability of GraphBLAS operations via fully functioning BFS and SSSP algorithms, as a proof of concept. We have also shown the advantage of a generic, modular design of our library and its adaptability through the hardware-specifically optimized backends switchable at compile-time. Our primary focus has not been performance, but the demonstration of an architecture that promotes scalability and performance while implementing the separation of concerns that allows for independent development of highly tuned backends.

Our current approach, however, leaves a lot to be desired in terms of both specification and performance. For example, having a structural zero different from the value "zero" (e.g., ∞) could be useful with certain specialized algorithms such as BFS with masking. We now only have experimental support for changeable structural zero values different from "zero" by having a dense matrix with all its structural values filled out by the specified zero value. Although equivalent in effect to a sparse-matrix-based approach, materializing and operating on a large dense matrix can be computationally expensive. Thus, finding a method to properly handle the assignment of structural zero values in a sparse setting is desirable. In addition, the performance of the current version of the library lags behind other GPU-specific graph primitives in literature, such as [16], by orders of magnitude in terms of runtime of the BFS algorithm on the same graph in equal settings.

We have identified some potential issues based on profiling results, and a thorough investigation and resolution of these issues is imminent. Some other questions also surfaced during our previous iteration of our implementation, such as whether to support vector data structure in our library and whether the vector should be sparse or dense. We are currently using a sparse matrix to approximate a sparse vector in our implementation, but using a sparse vector has several advantages over a matrix approximation or a dense vector, such as conservation of space and computation. There is, however, a trade-off in terms of memory cost when the sparse vector data structure actually represents a dense vector. These questions will need to be brought to the GraphBLAS community for further discussion.

As our next step, we plan to do the following:

- Participate in the ongoing discussion on the representation of structural zeros and its role with regard to specific GraphBLAS operations.
- Identify and characterize the performance bottlenecks and devise improvements while providing feedback to the third-party library developers.
- Explore additional features and specializations possible for operations, such as masking.
- Continue research and development of other graph algorithms using GraphBLAS operations.

VI. Acknowledgements

We thank the NVIDIA Corporation for donating two K40 GPUs to experiment with. We thank Steven Dalton from NVIDIA Research for help with running CUSP.

This material is based upon work funded and supported by the National Science Foundation under Grant No. 1111888 and by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

VII. References

[1] The Graph BLAS Forum, 2016. URL http://graphblas.org/.

[2] A. Buluc and J. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing*, 34(4): C170–C191, Jan. 2012. ISSN 1064-8275, 1095-7197. doi: 10.1137/110848244. URL http://arxiv.org/abs/1109.3739. arXiv: 1109.3739.

[3] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, Nov. 2011.

[4] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IEEE Internat. Symp. on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[5] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *28th International Parallel and Distributed Processing Symposium*, 2014.

[6] 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. Technical Report RC25508 (WAT1411-052), 2014.

[7] M. B. Enrico Mastrostefano. Efficient breadth first search on multi-GPU systems. *J. Parallel and Distributed Computing*, 73(9):1292–1305, 2013.

[8] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner. Graphulo: Linear algebra graph kernels for nosql databases. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 822–830. IEEE, 2015.

[9] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 267–276. ACM, 2011.

[10] J. Kepner. The GraphBLAS "Math document", Jan. 2016. URL http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf.

[11] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, Aug. 2011. ISBN 9780898719901.

[12] T. Mattson, D. Bader, J. Berry, A. Buluc, 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 *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, Sept. 2013. doi: 10.1109/HPEC.2013.6670338.

[13] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 117–128. ACM, 2012.

[14] L.-M. Munguia, D. A. Bader, and E. Ayguade. Task-Based Parallel Breadth-First Search in Heterogeneous Environments. In *Proc. 19th International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2012.

[15] Y. H. Timothy A. Davis. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1, 2011.

[16] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. *arXiv:1501.05387 [cs]*, Jan. 2015. URL http://arxiv.org/abs/1501.05387. arXiv: 1501.05387.

[17] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine. Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, pages 11:1–11:4. ACM, 2015.