GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU

CARL YANG, University of California, Davis and Lawrence Berkeley National Laboratory
AYDIN BULUÇ, Lawrence Berkeley National Laboratory and University of California, Berkeley
JOHN D. OWENS, University of California, Davis

ACM Reference Format:
Carl Yang, Aydın Buluç, and John D. Owens. 2021. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2021), 50 pages. https://doi.org/10.1145/3466795

High-performance implementations of graph algorithms are challenging to implement on new parallel hardware such as GPUs because of three challenges: (1) the difficulty of coming up with graph building blocks, (2) load imbalance on parallel hardware, and (3) graph problems having low arithmetic intensity. To address some of these challenges, GraphBLAS is an innovative, ongoing effort by the graph analytics community to propose building blocks based on sparse linear algebra, which allow graph algorithms to be expressed in a performant, succinct, composable, and portable manner. In this paper, we examine the performance challenges of a linear-algebra-based approach to building graph frameworks and describe new design principles for overcoming these bottlenecks. Among the new design principles is *exploiting input sparsity*, which allows users to write graph algorithms without specifying push and pull direction. *Exploiting output sparsity* allows users to tell the backend which values of the output in a single vectorized computation they do not want computed. *Load-balancing* is an important feature for balancing work amongst parallel workers. We describe the important load-balancing features for handling graphs with different characteristics. The design principles described in this paper have been implemented in "GraphBLAST", the first high-performance linear algebra-based graph framework on NVIDIA GPUs that is open-source. The results show that on a single GPU, GraphBLAST has on average at least an order of magnitude speedup over previous GraphBLAS implementations SuiteSparse and GBTL, comparable performance to the fastest GPU hardwired primitives and shared-memory graph frameworks Ligra and Gunrock, and better performance than any other GPU graph framework, while offering a simpler and more concise programming model.

Authors' addresses: Carl Yang, University of California, Davis and Lawrence Berkeley National Laboratory, Dept. of Electrical and Computer Engineering, 1 Shields Avenue, Davis, California, 95616, ctcyang@ucdavis.edu; Aydın Buluç, Lawrence Berkeley National Laboratory and University of California, Berkeley, 1 Cyclotron Road, Berkeley, California, 94720, abuluc@lbl.gov; John D. Owens, University of California, Davis, Dept. of Electrical and Computer Engineering, 1 Shields Avenue, Davis, California, 95616, jowens@ece.ucdavis.edu.

0098-3500/2021/1-ART1
https://doi.org/10.1145/3466795

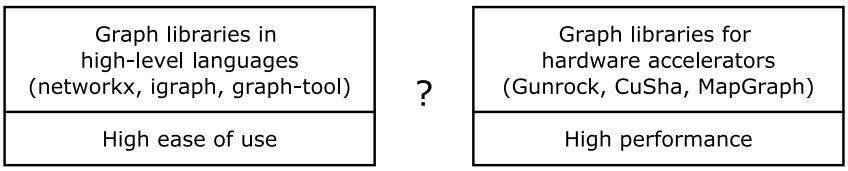


Fig. 1. Mismatch between existing frameworks targeting high-level languages and hardware accelerators.

1 INTRODUCTION

Graphs are a representation that naturally emerges when solving problems in domains including bioinformatics [38], social network analysis [21], molecular synthesis [45], and route planning [30]. Graphs may contain billions of vertices and edges, so parallelization has become a must.

The past two decades have seen the rise of parallel processors into a commodity product—both general-purpose processors in the form of graphic processor units (GPUs), as well as domain-specific processors such as tensor processor units (TPUs) and the graph processors being developed under the DARPA HIVE program. Research into developing parallel hardware and initiatives such as the DIMACS and HPEC graph challenges [47, 70] have succeeded in speeding up graph algorithms [76, 82]. However, the improvement in graph performance has come at the cost of a more challenging programming model. The result has been a mismatch between the high-level languages that users and graph algorithm designers would prefer to program in (e.g., Python) and programming languages for parallel hardware (e.g., C++, CUDA, OpenMP, or MPI).

To address this mismatch, many initiatives, including NVIDIA's RAPIDS effort [65], have been launched in order to provide an open-source Python-based ecosystem for data science and graphs on GPUs. One such initiative, GraphBLAS, is an open standard [17] for graph frameworks. It promises standard building blocks for expressing graph algorithms in the language of linear algebra. Such a standard attempts to solve the following problems:

(1) *Performance portability*: Graph algorithms need no modification to have high performance across hardware.
(2) *Concise expression*: Graph algorithms are expressed in few lines of code.
(3) *High-performance*: Graph algorithms achieve state-of-the-art performance.
(4) *Scalability*: An implementation is effective at both small-scale and exascale.

Goal 1 (*performance portability*) is central to the GraphBLAS philosophy, and it has made inroads with several implementations already being developed using this common interface [27, 61, 88]. Regarding Goal 2 (*concise expression*), GraphBLAS encourages users to think in a vectorized manner, which yields an order-of-magnitude reduction in lines of code as shown in Table 1. Before Goal 4 (*scalability*) can be achieved, Goal 3 (*high-performance*) on the small scale must first be demonstrated.

However to date, GraphBLAS has lacked high-performance implementations for GPUs. The GraphBLAS Template Library (GBTL) [88] is a GraphBLAS-inspired GPU graph framework. The architecture of GBTL is C++-based and maintains a separation of concerns between a top-level interface defined by the GraphBLAS C API specification and the low-level backend. However, since it was intended as a proof-of-concept in programming language research, it is an order of magnitude slower than state-of-the-art graph frameworks on the GPU in terms of performance.

We identify several reasons graph frameworks are *challenging* to implement on the GPU:

Generalizability of optimization While many graph algorithms share similarities, the optimizations found in high-performance graph frameworks often seem ad hoc and difficult to reconcile with the goal of a clean and simple interface. What are the *optimizations* most deserving of attention when designing a high-performance graph framework on the GPU?

Algorithm	This Work	Framework CS	GL	GR	LI	MG	GB	SS
Breadth-first-search	22	76	353	1161	45	140	22	29
Single-source shortest-path	24	78	440	465	60	184	25	N/A
PageRank	32	84	342	805	68	144	47	31
Connected components	50	N/A	595	1435	61	153	N/A	132
Triangle counting	8	N/A	283	297	60	N/A	17	15

Table 1. Comparison of lines of C or C++ application code for seven graph frameworks and this work. The graph frameworks we compared with are CuSha (CS) [51], Galois (GL) [62], Gunrock (GR) [82], Ligra (LI) [76], Mapgraph (MG) [37], GBTL (GB) [88], and SuiteSparse (SS) [27].

Load imbalance Graph problems have irregular memory access patterns making it hard to extract parallelism. On parallel systems such as GPUs, this is further complicated by the challenge of balancing work amongst parallel compute units. How should this problem of *load-balancing* be addressed?

Low compute-to-memory access ratio Graph problems typically require multiple memory accesses on unstructured data rather than many floating-point computations. Therefore, graph problems are often memory-bound rather than compute-bound. What can be done to reduce the *number of memory accesses*?

What are the design principles required to build a GPU implementation based on linear algebra that matches the state-of-the-art graph frameworks in performance? Towards that end, we have designed GraphBLAST[1]: the first high-performance implementation of GraphBLAS for the GPU. Our implementation is for a single GPU, but given the similarity between the GraphBLAS interface we are adhering to and the CombBLAS interface [15], which is a graph framework for distributed CPUs, we are confident the design we propose here will allow us to extend it to a distributed implementation with future work.

In order to perform a comprehensive evaluation of our system, we compare our framework against state-of-the-art graph frameworks on the CPU and GPU, as well as hardwired GPU implementations, which are problem-specific GPU implementations that developers have hand-tuned for performance. The state-of-the-art graph frameworks against which we will be comparing are Ligra [76], Gunrock [82], CuSha [51], Galois [62], Mapgraph [37], GBTL [88], and SuiteSparse [27], which we will describe in greater detail in Section 2.2. The hardwired implementations will be Enterprise (BFS) [55], delta-stepping SSSP [26], pull-based PR [51], hooking and pointer-jumping CC [79], and bitmap-based triangle counting [12]. The five graph algorithms on which we will be evaluating our system are:

- Breadth-first-search (BFS)
- Single-source shortest-path (SSSP)
- PageRank (PR)
- Connected components (CC)
- Triangle counting (TC)

GraphBLAST has also been used for graph coloring [63] as well as DARPA HIVE graph algorithms on the GPU [46], including graph projections, local graph clustering, and seeded graph matching.

Our contributions in this paper are as follows:

[1]https://github.com/gunrock/graphblast

Major Feature	Component	Application BFS	SSSP	PR	CC	TC
Exploit input sparsity	Generalized direction optimization	✓	✓	✓	✓	
	Boolean semiring	✓				
	Avoid sparse-to-dense conversion	✓	✓			
Exploit output sparsity	Masking	✓	✓		✓	✓
Load-balancing	Row split	✓	✓	✓	✓	✓
	Merge-based	✓	✓	✓	✓	

Table 2. Applicability of design principles.

(1) We briefly categorize parallel graph frameworks (Section 2) and give a short introduction to GraphBLAS's computation model (Section 3).
(2) We demonstrate the importance of exploiting *input sparsity*, which means picking the algorithm based on a cost model that selects between an algorithm that exploits the input vector's sparsity and another algorithm that is more efficient for denser input vectors. One of the consequences is direction optimization (Section 4).
(3) We show the importance of exploiting *output sparsity*, which is implemented as masking and can be used to reduce the number of memory accesses of several graph algorithms (Section 5).
(4) We explain the design considerations required for high-performance on the GPU, which are avoiding CPU-to-GPU memory copies, supporting generalized semiring operators, and load-balancing (Section 6).
(5) We review how common graph algorithms are expressed in GraphBLAST (Section 7).
(6) We show that, enabled by the optimizations *exploiting sparsity*, *masking*, and *proper load-balancing*, GraphBLAST gets 43.51× geomean (i.e., geometric mean) and 1268× peak over SuiteSparse GraphBLAS for multi-threaded CPUs. Compared to state-of-the-art graph frameworks on the CPU and GPU on five graph algorithms running on scale-free graphs, GraphBLAST gets 2.31× geomean (10.97× peak) and 1.14× (5.24× peak) speed-up (Section 8).

Over the next three sections, we will discuss the most important design principles for making this code performant, which are exploiting input sparsity and output sparsity, and making good decisions for considerations specific to the GPU. Table 2 shows which of the five graph algorithms discussed in this paper our optimizations apply to.

2 BACKGROUND & MOTIVATION

We begin by describing related literature in the field of graph frameworks on parallel hardware (Section 2.1), and move to discussing the limitations of previous systems that inspired ours (Section 2.2). Further, we review the connection between graph algorithms and linear algebra (Section 2.3). For a broader survey of parallel graph frameworks, refer to Doekemeijer and Varbanescu's 2014 work [32].

2.1 Related work

Large-scale graph frameworks on multi-threaded CPUs, distributed-memory CPU systems (surveyed by Batarfi et al. [3]), and massively parallel GPUs (surveyed by Shi et al. [74]) fall into three broad categories: vertex-centric (surveyed by McCune, Weninger and Madey [58]), edge-centric,

and linear-algebra-based. In this section, we will explain this categorization and the influential graph frameworks from each category.

2.1.1 Vertex-centric. Introduced by Pregel [56], vertex-centric frameworks are based on parallelizing over vertices. The computation in Pregel is inspired by the distributed CPU programming model of MapReduce [29] and is based on message passing. At the beginning of the algorithm, all vertices are active. Pregel follows the bulk synchronous programming model (BSP) consisting of global synchronization barriers called *supersteps*. At the end of a superstep, the runtime receives the messages from each sending vertex and computes the set of active vertices for the superstep. Computation continues until convergence or a user-defined condition is reached.

Pregel's programming model is good for scalability and fault tolerance. However, standard graph algorithms in most Pregel-like graph processing systems suffer from slow convergence on large-diameter graphs and load imbalance on scale-free graphs. Apache Giraph [21] is an open-source implementation of Google's Pregel. It is a popular graph computation engine in the Hadoop ecosystem initially open-sourced by Yahoo!. Han et al. [42] provide a full survey of Pregel-like frameworks.

Galois [62] is a graph system for shared memory based on a different operator abstraction that supports priority scheduling and dynamic graphs and processes on subsets of vertices called active elements. However, their model does not abstract implementation details of the loop from the user. Users have to generate the active elements set directly for different graph algorithms.

First introduced by PowerGraph [40], the Gather-Apply-Scatter (GAS) model is a concrete implementation of the vertex-centric model designed to address the slow convergence of vertex-centric models on power law graphs. For the load imbalance problem, it uses vertex-cut to split high-degree vertices into equal degree-sized redundant vertices. This exposes greater parallelism in real-world graphs. It supports both BSP and asynchronous execution. Like Pregel, PowerGraph is a distributed CPU framework. For flexibility, PowerGraph also offers a vertex-centric programming model, which is efficient on non-power law graphs.

MapGraph [37] is a similar GAS framework and integrates both Baxter's load-balanced search [4] and Merrill, Garland, and Grimshaw's dynamic grouping workload mapping strategy [60] to increase its performance. CuSha [51] is also a GAS model-based GPU graph analytics system. It solves the load imbalance and GPU underutilization problem with a GPU adoption of the parallel sliding window technique. They call this preprocessing step "G-Shard" and combine it with a concatenated window method to group edges from the same source indices.

Noteworthy systems for processing dynamic graphs are STINGER [34], Hornet [19], Kineograph [20], Aspen [31], and Terrace [64]. The systems all avoid using the popular CSR data structure for storing the graph. For example, Hornet stores the adjacency list in arrays of memory blocks, using a vectorized bit tree to find the next available memory block, and leveraging B+ trees for managing memory blocks [5].

HavoqGT is a distributed graph framework built for high performance [66, 67]. Its novelty is a new algorithmic technique called *vertex delegates* in its programming model, which both load-balances and performs asynchronous broadcast and reduction operations for the high-degree vertices. This has the impact of performing much better than a simple 1D partitioning strategy on distributed systems.

2.1.2 Edge-centric. First introduced by X-Stream [69], the edge-centric model treats edges rather than vertices as the first-class graph entities. There, authors build an out-of-core engine that relies on streaming unordered edge lists. Even though updates to vertices must still be random access, the updates to edges can have sequential access. Roy et al. [69] contend that this takes advantage

of storage media (main memory, solid-state disk, and magnetic disk) having superior sequential access performance vs. random memory access.

2.1.3 Linear algebra-based. Linear algebra-based graph frameworks were pioneered by the Combinatorial BLAS (CombBLAS) [15], a distributed memory CPU-based graph framework. Algebra-based graph frameworks rely on the fact that graph traversal can be described as a matrix-vector product. CombBLAS offers a small but powerful set of linear algebra primitives. Combined with algebraic semirings, this small set of primitives can describe a broad set of graph algorithms. The advantage of CombBLAS is that it is the only framework that can express a 2D partitioning of the adjacency matrix, which is helpful in scaling to large-scale graphs.

In the context of bridging the gap between vertex-centric and linear algebra-based frameworks, GraphMat [80] is a groundbreaking work. Traditionally, linear algebra-based frameworks have found difficulty gaining adoption, because they rely on users' understanding how to express graph algorithms in terms of linear algebra. GraphMat addresses this problem by exposing a vertex-centric interface to the user, automatically converting such a program to a generalized sparse matrix-vector multiply, and then performing the computation on a linear-algebra-based backend.

nvGRAPH [33] is a high-performance GPU graph analytics library developed by NVIDIA. It views graph analytics problems from the perspective of linear algebra and matrix computations [50], and uses semiring matrix-vector multiply operations to present graph algorithms. As of version 10.1, it supports five algorithms: PageRank, single-source shortest-path (SSSP), triangle counting, single-source widest-path, and spectral clustering. SuiteSparse [27] is notable for being the first GraphBLAS-compliant library. We compare against the multithreaded CPU implementation of SuiteSparse. GBTL [88] is a GraphBLAS-like framework on the GPU. Rather than high performance, its implementation focused on programming language research and a separation of concerns between the interface and the backend.

2.1.4 Implementation challenges on GPUs. Whether vertex-centric, edge-centric, or linear-algebra-based, GPU implementations of graph frameworks face several common challenges in achieving high performance.

Fine-grained load imbalance. The most straightforward form of parallelism in graph problems is parallelizing across vertices. However, in many graphs, particularly scale-free graphs, the number of outbound edges at each vertex may vary dramatically. Consequently, the amount of work per vertex varies in the same way. Thus, a GPU implementation that assigns vertices to neighboring threads results in significant fine-grained load imbalance across those neighboring threads. This imbalance is identical to the imbalance in sparse matrix operations with a variable amount of non-zero elements per row that choose to assign a thread per matrix row [9].

We describe this problem from the linear algebra perspective in Section 6. Native graph frameworks typically address this problem through a variety of techniques, including dynamically binning vertices by the size of their workload and processing like-sized bins with an appropriately sized grain of computation [60] or converting parallelism over vertices to parallelism over edges using prefix-sum-like methods [26]. The challenges are choosing the right load-balance method and balancing the cost of load balance vs. its performance benefits.

Minimizing overhead. GPU kernels that run on large, load-balanced datasets with a large amount of work per input element achieve their peak throughput. However, in the course of a graph computation, a GPU framework may often face situations where its runtime is not dominated by processing time but instead by overheads. One form of overhead occurs when the GPU does not have enough work to keep the entire GPU busy; in such a case, that kernel's runtime is dominated by the cost of the kernel launch rather than the cost of the work. A related form of overhead for

bulk-synchronous operations is the requirement for global synchronization at the end of each kernel; to first order, the entire GPU must wait until the last element processed by a kernel is complete before starting the next kernel.

Another form of overhead is when multiple kernels are used to perform a computation that could be combined into a single kernel ("kernel fusion"), one that potentially exploits producer-consumer locality within a kernel. The performance gap between "hardwired" graph algorithm implementations that are tailored to a single algorithm and more general programmable graph frameworks is often a result of this additional overhead for the programmable framework [82, Section 3.1] because frameworks are generally built from smaller, modular operations and cannot automatically perform kernel fusion and exploit producer-consumer locality as hardwired implementations do.

The overhead of a kernel launch is on the order of several microseconds [87]. So, for graph computations that require many iterations and have many kernel launches per iteration, the aggregate cost of kernel launches is significant. Thus, minimizing kernel launches is an important goal of any high-performance graph framework.

2.2 Previous systems

Two systems that directly inspired our work are Gunrock and Ligra.

2.2.1 Gunrock. Gunrock [82] is a state-of-the-art GPU-based graph processing framework. It is notable for being the only high-level GPU-based graph analytics system with support for both vertex-centric and edge-centric operations, as well as fine-grained runtime load balancing strategies, without requiring any preprocessing of input datasets. Indeed, Table 3 shows Gunrock has the most performance optimizations out of all graph frameworks, but this comes at a cost of increasing the complexity and amount of user application code. In our work, we want the performance Gunrock optimizations provide while moving more work to the backend. In other words, we want to adhere to GraphBLAS's compact and easy-to-use user interface, while maintaining state-of-the-art performance.

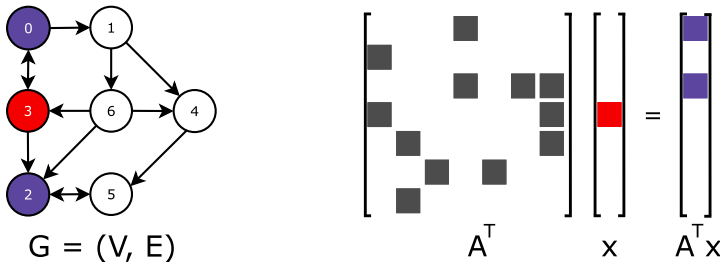
2.2.2 Ligra. Ligra [76] is a CPU-based graph processing framework for shared memory. Its lightweight implementation is targeted at shared memory architectures and uses CilkPlus for its multithreading implementation. It is notable for being the first graph processing framework to generalize Beamer, Asanović and Patterson's direction-optimized BFS [7] to many graph-traversal-based algorithms. However, Ligra does not support multi-source graph traversals. In our framework, multi-source graph traversals find natural expression as sparse BLAS 3 operations (matrix-matrix multiplications).

2.3 Graph traversal vs. matrix-vector multiply

The connection between graph traversal and linear algebra was noted by Denes König [54] in the early days of graph theory. Since then the connection between graphs and matrices has been established by the popular representation of a graph as an adjacency matrix. More specifically, it has become popular to represent a vector-matrix multiply as being equivalent to one iteration of breadth-first-search traversal (see Figure 2). Some seemingly non-traversal graph algorithms such as triangle counting, which can be solved efficiently using a masked SpGEMM, can also be thought in terms of traversals. Multiplying the lower triangle of the adjacency matrix with its transpose, as we do in Section 7.5, simply does a 2-hop traversal of the graph from every vertex. The use of lower triangle and its transpose as opposed to the whole adjacency matrix ensures that identical paths are not explored redundantly. The masking checks whether such 2-hop paths, also called *wedges* or *triads* in the literature, close to form triangles.

Component	This Work	Framework CS	GL	GR	LI	MG	GB	SS
Programming model	LA	GA	GA	GA	GA	GA	LA	LA
Backend	GPU	GPU	CPU	GPU	CPU	GPU	GPU	CPU
Preprocessing	no	yes	no	no	no	no	no	no
BFS lines of code	22	76	353	1161	45	140	22	29
Direction optimization	✓		✓	✓	✓			✓
Generalized direction optimization	✓				✓			
Early-exit optimization [85]	✓			✓				✓
Structure-only optimization [85]	✓	✓	✓	✓	✓	✓	✓	✓
Avoid sparse-to-dense conversion [85]	✓			✓				
Masking (kernel fusion)	✓			✓				✓
Static mapping (vertex-centric)	✓	✓	✓	✓	✓	✓	✓	✓
Dynamic mapping (edge-centric)	✓			✓	✓			✓

Table 3. Detailed comparison of different parallel graph frameworks on the CPU and GPU. LA indicates a linear algebra-based model and GA indicates a native graph abstraction composed of vertices and edges. The five graph abstraction-based frameworks we compared with are CuSha (CS) [51], Galois (GL) [62], Gunrock (GR) [82], Ligra (LI) [76], and MapGraph (MG) [37]. The two linear-algebra-based frameworks we compared with are GBTL (GB) [88] and SuiteSparse (SS) [27]. Note that part of load balancing work in CuSha is done during the (offline) G-shard generation process. The difference between direction optimization and generalized direction optimization is that the former indicates the framework supports this optimization, while the latter indicates the selection of push and pull is automated and generalized to graph algorithms besides BFS. Early-exit, structure-only and avoid sparse-to-dense conversion optimizations are discussed in previous work [85].



(a) graph traversal (b) matrix-vector multiply

Fig. 2. The adjacency matrix $\mathbf{A}$ is one representation of graph $G = (V, E)$ with its set of vertices V and set of edges E. The matrix-vector multiply $\mathbf{A}^\mathsf{T}\mathbf{x}$ is one representation of the BFS graph traversal where the sparse vector $\mathbf{x}$ represents the current active frontier of vertices.

3 GRAPHBLAS CONCEPTS

The following section introduces GraphBLAS's model of computation. A full treatment of GraphBLAS is beyond the scope of this paper; we give a brief introduction to the reader, so that he or she can better follow our contributions in later sections. We refer the interested reader to the GraphBLAS C API specification [17] and selected papers [18, 50, 57] for a full treatment. At the end of this section, we give a running example (Section 3.4). In later sections, we will show how taking

Operation	Description	Graph application
`Matrix`	matrix constructor	create graph
`Vector`	vector constructor	create vertex set
`dup`	copy assignment	copy graph or vertex set
`clear`	empty vector or matrix	empty graph or vertex set
`size`	no. of elements (vector only)	no. of vertices
`nrows`	no. of rows (matrix only)	no. of vertices
`ncols`	no. of columns (matrix only)	no. of vertices
`nvals`	no. of stored elements	no. of active vertices or edges
`build`	build sparse vector or matrix	build vertex set or graph from tuples
`buildDense`†	build dense vector or matrix	build vertex set or graph from tuples
`fill`†	build dense vector or matrix	build vertex set or graph from constant
`setElement`	set single element	modify single vertex or edge
`extractElement`	extract single element	read value of single vertex or edge
`extractTuples`	extract tuples	read values of vertices or edges

Table 4. A list of Matrix and Vector operations in GraphBLAST.
†: These are convenience operations not found in the GraphBLAS specification, but were added by the authors for GraphBLAST.

advantage of *input* and *output sparsity* will, even in the small running example, allow computation to complete with fewer memory accesses.

GraphBLAS's model of computation includes the following concepts:

(1) Abstract algebraic constructs: Matrix, Vector, Semiring, and Monoid
(2) Programming constructs: Masking and Descriptor
(3) Compute using constructs: Operation

As the name may suggest, the abstract algebraic constructs (Matrix, Vector, Semiring, and Monoid) come directly from abstract algebra and have precise mathematical definitions from that community. The programming constructs (Masking and Descriptor) are used to provide expressibility and performance by changing the abstract algebraic constructs or operations slightly. The Operations are the functions used to carry out computation over the algebraic and programming constructs.

3.1 Abstract algebraic constructs

3.1.1 Matrix. A Matrix in GraphBLAST is a general M-by-N matrix but it is often used to represent the adjacency matrix of a graph. A full list of methods used to interact with Matrix objects is shown in Table 7. When referring to matrices in mathematical notation, we will indicate them with uppercase boldface, i.e., $\mathbf{A}$. We index the (i, j)-th element of the matrix with $\mathbf{A}(i, j)$, the i-th row with $\mathbf{A}(i, :)$, and the j-th column with $\mathbf{A}(:, j)$.

3.1.2 Vector. A Vector is the set of vertices in a graph that are currently actively involved in the graph computation. We call these vertices *active*. The list of methods used to interact with Vector objects overlaps heavily with the one for Matrix objects. When referring to vectors in mathematical notation, we will indicate them with lowercase boldface, i.e., $\mathbf{x}$. We index the i-th element of the vector with $\mathbf{x}(i)$.

Name	Semiring	Application
PlusMultiplies	$\{+, \times, \mathbb{R}, 0\}$	Classical linear algebra
LogicalOrAnd	$\{\|\|, \&\&, \{0, 1\}, 0\}$	Graph connectivity
MinPlus	$\{\min, +, \mathbb{R} \cup \{+\infty\}, +\infty\}$	Shortest path
MaxPlus	$\{\max, +, \mathbb{R}, -\infty\}$	Graph matching
MinMultiplies	$\{\min, \times, \mathbb{R}, +\infty\}$	Maximal independent set
Name	**Monoid**	**Application**
PlusMonoid	$\{+, 0\}$	Sum-reduce
MultipliesMonoid	$\{\times, 1\}$	Times-reduce
MinimumMonoid	$\{\min, +\infty\}$	Min-reduce
MaximumMonoid	$\{\max, -\infty\}$	Max-reduce
LogicalOrMonoid	$\{\|\|, 0\}$	Or-reduce
LogicalAndMonoid	$\{\&\&, 1\}$	And-reduce

Table 5. A list of commonly used semirings and monoids in GraphBLAST.

3.1.3 Semiring. A semiring encapsulates computation on vertices and edges of the graph. In classical matrix multiplication, the semiring used is the $(+, \times, \mathbb{R}, 0)$ arithmetic semiring. However, this can be generalized to $(\oplus, \otimes, \mathbb{D}, \mathbb{I})$ in order to vary what operations are performed during the graph search. $(\oplus, \otimes, \mathbb{D}, \mathbb{I})$ represents the following:

- $\otimes$: Semiring multiply
- $\oplus$: Semiring add
- $\mathbb{D}$: Semiring domain
- $\mathbb{I}$: Additive identity

Here is an example using the MinPlus semiring (also known as the tropical semiring) $(\oplus, \otimes, \mathbb{D}, \mathbb{I}) = \{\min, +, \mathbb{R} \cup \{+\infty\}, +\infty\}$, which can be used for shortest-path calculation:

- $\otimes$: In MinPlus, $\otimes = +$. The vector represents currently known shortest distances between a source vertex s and vertices whose distance from s we want to update, say v. During the multiplication $\otimes = +$, we want to add up distances from parents of v whose distance from s is finite. This gives distances from $s \rightarrow u \rightarrow v$, potentially via many parent vertices u.
- $\oplus$: In MinPlus, $\oplus = \min$. This operation chooses choosing the distance from $s \rightarrow u \rightarrow v$ such that the distance is a minimum for all intermediate vertices u.
- $\mathbb{D}$: In MinPlus, $\mathbb{D} = \mathbb{R} \cup \{+\infty\}$, which is the set of real numbers augmented by infinity (indicating unreachability).
- $\mathbb{I}$: In MinPlus, $\mathbb{I} = +\infty$, representing that doing the reduction $\oplus$ if there are no elements to be reduced—there is no parent u that is reachable from s—the default output should be infinity, indicating v is unreachable from s as well.

The most frequently used semirings are shown in Table 5.

3.1.4 Monoid. A monoid is similar to a semiring, but it only has one operation, which must be associative and have an identity. A monoid should be passed in to GraphBLAS operations that only need one operation instead of two. As a rule of thumb, the only operations that require two operations (i.e., a semiring) are mxm, mxv, and vxm. This means that for GraphBLAS operations eWiseMult, eWiseAdd, and reduce, a monoid should be passed in. A list of frequently used monoids is shown in Table 5.

Field	Value	Behavior
GrB_MASK	(default)	Mask
	GrB_SCMP	Structural complement of mask
GrB_INP0	(default)	Do not transpose first input parameter
	GrB_TRAN	Transpose first input parameter
GrB_INP1	(default)	Do not transpose second input parameter
	GrB_TRAN	Transpose second input parameter
GrB_OUTP	(default)	Do not clear output before writing to masked indices
	GrB_REPLACE	Clear output before writing to masked indices

Table 6. A list of descriptor settings in GraphBLAST. Below the line are variants that are in the GraphBLAS API specification that we do not currently support.

3.2 Programing constructs

3.2.1 Masking. Masking is an important tool in GraphBLAST that lets a user mark the indices where the result of any operation in Table 7 should be written to the output. This set of indices is called the *mask* and must be in the form of a `Vector` or `Matrix` object. The masking semantic is:

> *For a given pair of indices* (i, j)*, if the mask matrix* $\mathbf{M}(i, j)$ *has a value 0, then the output at location* (i, j) *will not be written to* $\mathbf{C}(i, j)$*. However, if* $\mathbf{M}(i, j)$ *is not equal to 0, then the output at location* (i, j) *will be written to* $\mathbf{C}(i, j)$*.*

Sometimes, the user may want the opposite to happen: when the mask matrix has a value 0 at $\mathbf{M}(i, j)$, then it will be written to the output matrix $\mathbf{C}(i, j)$. Likewise, if the mask matrix has a non-zero, then it will not be written. This construction is called the *structural complement* of the mask.

To represent masking, we borrow the elementwise multiplication operation from MATLAB. Given we are trying to multiply matrices $\mathbf{A}$, $\mathbf{B}$ into matrix $\mathbf{C}$ and the operation is masked by matrix $\mathbf{M}$, our operation is $\mathbf{C} \leftarrow \mathbf{AB} .* \mathbf{M}$.

3.2.2 Descriptor. A descriptor is an object passed into all operations listed in Table 7 that can be used to modify the operation. For example, a mask can be set to use the structural complement using a method `Descriptor::set(GrB_MASK, GrB_SCMP)`. The other operations we include are listed in Table 6.

3.3 Operation

An operation is a commonly used linear algebra operation. A full list of operations is shown in Table 7. Of the operations in Table 7, the most computationally intensive and useful operations are `mxm`, `mxv`, and `vxm`. We find empirically that these operations take over 90% of graph algorithm runtime. For these operations, we decompose them into constituent parts in order to better optimize their performance (see Figure 4). Intuitively, `mxv` and `vxm` represent a graph traversal from a single-nodeset (one single set of active nodes $\mathbf{u}(i)$ for all i). On ther other hand, `mxm` represents a multi-nodeset, that is to say, a graph-traversal from multiple, independent sets of active nodes at the same time (for all independent sets of active nodes $B(:, j)$, start a traversal for all active nodes i in that set).

An operation is used to tie all the objects in GraphBLAS together. Figure 3 shows a matrix-vector multiply, which represents a graph traversal from all active nodes $\mathbf{u}(i)$ to their neighbor nodes, applying the semiring multiply $\otimes$ to get a temporary $\mathbf{c}(i, j) = \mathbf{A}(i, j) \otimes \mathbf{u}(i)$ between the edge value

Operation	Math Equivalent	Description	Graph application
`mxm`	$\mathbf{C} = \mathbf{AB}$	matrix-matrix mult.	multi-nodeset traversal
`mxv`	$\mathbf{w} = \mathbf{Au}$	matrix-vector mult.	single-nodeset traversal
`vxm`	$\mathbf{w} = \mathbf{uA}$	vector-matrix mult.	single-nodeset traversal
`eWiseMult`	$\mathbf{C} = \mathbf{A} .* \mathbf{B}$	element-wise mult.	graph intersection
	$\mathbf{w} = \mathbf{u} .* \mathbf{v}$		vertex intersection
`eWiseAdd`	$\mathbf{C} = \mathbf{A} + \mathbf{B}$	element-wise add	graph union
	$\mathbf{w} = \mathbf{u} + \mathbf{v}$		vertex union
`extract`	$\mathbf{C} = \mathbf{A}(\mathbf{i}, \mathbf{j})$	extract submatrix	extract subgraph
	$\mathbf{w} = \mathbf{u}((i))$	extract subvector	extract subset of vertices
`assign`	$\mathbf{C}(\mathbf{i}, \mathbf{j}) = \mathbf{A}$	assign to submatrix	assign to subgraph
	$\mathbf{w}(\mathbf{i}) = \mathbf{u}$	assign to subvector	assign to subset of vertices
`apply`	$\mathbf{C} = f(\mathbf{A})$	apply unary op	apply function to each edge
	$\mathbf{w} = f(\mathbf{u})$		apply function to each vertex
`reduce`	$\mathbf{w} = \sum_i \mathbf{A}(i, :)$	reduce to vector	compute out-degrees
	$\mathbf{w} = \sum_j \mathbf{A}(:, j)$	reduce to vector	compute in-degrees
	$w = \sum \mathbf{w}$	reduce to scalar	
`transpose`	$\mathbf{C} = \mathbf{A}^T$	transpose	reverse edges in graph

Table 7. A list of operations in GraphBLAST.

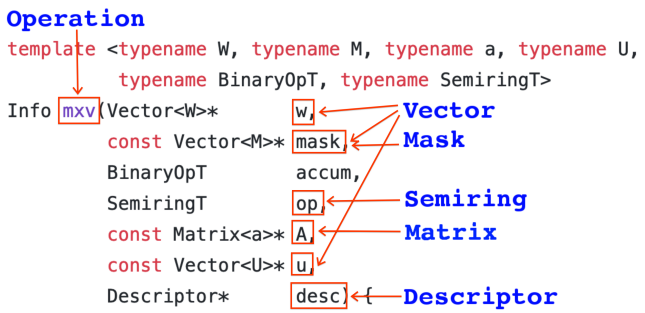


Fig. 3. Calling the operation `mxv` (matrix-vector multiply) performs $\mathbf{w} = \mathbf{Au}. * \mathbf{mask}$ over the Semiring op. The template parameters can be used to do compile-time type-checking. `Info` is an error type that is returned according to the C API specification [17]. `accum` is an optional parameter for controlling whether the output of the calculation overwrites $\mathbf{w}$ or whether it is accumulated to $\mathbf{w}$. The Descriptor can be used to control whether the matrix should be transposed or not, and whether the mask or the structural complement of the mask is used (see Section 3.2.1).

$\mathbf{A}(i, j)$ and the active node's value $\mathbf{u}(i)$ and finally doing a reduction over the temporary output $\mathbf{w}(i) = \bigoplus_j \mathbf{c}(i, j)$ using the semiring add operation $\oplus$ and the semiring identity I. Optionally, the user has the option of applying a boolean mask to the output so that the output is effectively $\mathbf{w}(i) = \mathbf{mask}(i) \times \mathbf{w}(i)$ for all i. The descriptor can further mutate this operation to account for matrix transposition and whether the mask or its structural complement should be used.

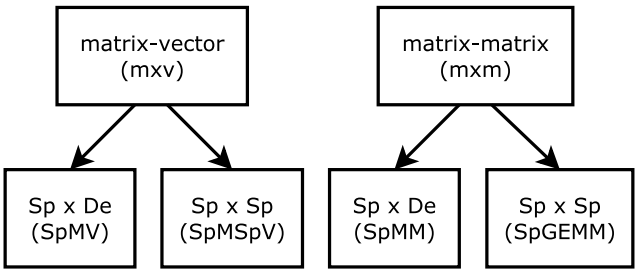


Fig. 4. Decomposition of key GraphBLAS operations. Note that vxm is the same as mxv and setting the matrix to be transposed, so it is not shown.

3.4 Running example

As a running example in this paper, we discuss sparse-matrix multiplication by dense-vector (SpMV) and sparse-vector (SpMSpV) with a direct dependence on graph traversal. The key step we will be discussing is Line 8 of Algorithm 1, which is the matrix formulation of parallel breadth-first-search. Illustrated in Figure 5b, this problem consists of a matrix-vector multiply followed by an elementwise multiplication between two vectors: one is the output of the matrix-vector multiply and the other is the negation (or *structural complement*) of the visited vector.

Using the standard dense matrix-vector multiplication algorithm (GEMV), we would require $8 \times 8 = 64$ loads and store instructions. However, if we instead treat the matrix not as a dense matrix but as a sparse matrix in order to take advantage of input matrix sparsity, we can perform the same computation in a mere 20 loads and stores into the sparse matrix. This number comes from counting the number of nonzeroes in the sparse matrix, which is equivalent to the number of edges in the graph. Using this as the baseline, we will show in later sections how optimizations such as exploiting the input vector and output vector sparsity can further reduce the number of loads and stores required.

3.5 Code example

Having described the different components of GraphBLAST, we show a short code example of how to do breadth-first-search using the GraphBLAST interface alongside the linear algebra in Algorithm 1. Before the while-loop, the vectors $\mathbf{f}$ and $\mathbf{v}$ (representing the vertices currently active in the traversal and the set of previously visited vertices) are initialized.

Then in each iteration of the while-loop, the following steps take place: (1) vertices currently active are added to the visited vertex vector, marked by the iteration d where they were first encountered; (2) the active vertices are traversed to find the next set of active vertices, and then elementwise-multiplied by the negation of the set of active vertices (filtering out previously visited vertices); (3) the number of active vertices of the next iterations is reduced to variable c; (4) the iteration number is incremented. This while-loop continues until there are no more active vertices (c reaches 0).

As demonstrated in the code example, GraphBLAS has the advantage of being concise. Developing new graph algorithms in GraphBLAS requires modifying a single file and writing simple C++ code. Provided a GraphBLAS implementation exists for a particular processor, GraphBLAS code can be used with minimal changes. Over the next three sections, we will discuss the most important design principles for making this code performant: exploiting input sparsity, exploiting output sparsity, and good load-balancing.

procedure MatrixBFS(Graph $\mathbf{A}$, Vector $\mathbf{v}$, Source s)
  Initialize $d \leftarrow 1$
  Initialize $\mathbf{f}(i) \leftarrow \begin{cases} 1, & \text{if } i = s \\ 0, & \text{if } i \neq s \end{cases}$
  Initialize $\mathbf{v} \leftarrow [0, 0, \ldots, 0]$
  Initialize $c \leftarrow 1$
  while $c > 0$ **do**
    Update $\mathbf{v} \leftarrow d\mathbf{f} + \mathbf{v}$
    Update $\mathbf{f} \leftarrow \mathbf{A}^{\mathrm{T}}\mathbf{f} .* \neg\mathbf{v}$ ▷ using Boolean semiring (see Table 5)
    Compute $c \leftarrow \sum_{i=0}^{n} \mathbf{f}(i)$ ▷ using standard plus monoid (see Table 5)
    Update $d \leftarrow d + 1$
  end while
end procedure

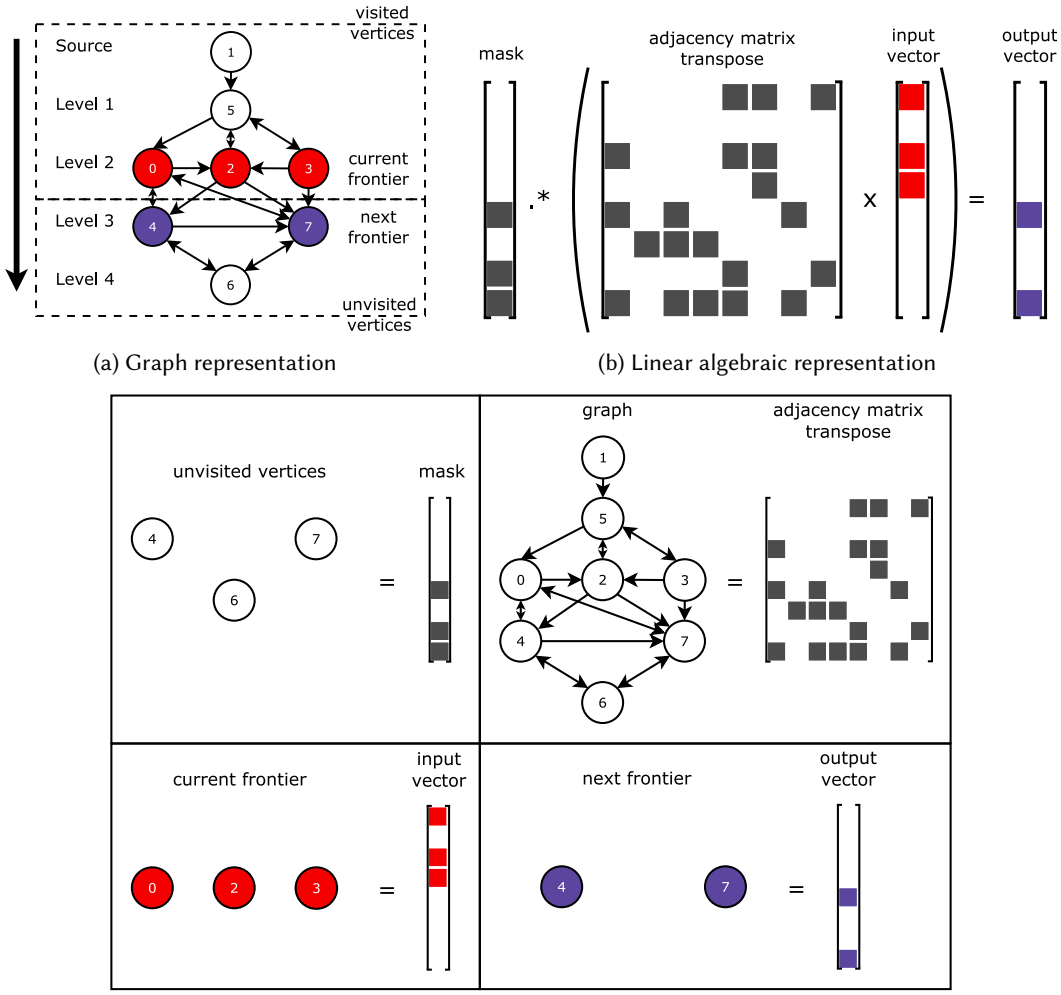
```cpp
#include <graphblas/graphblas.hpp>

void bfs(Vector<float>*       v,
         const Matrix<float>* A,
         Index                s) {
  Descriptor desc;
  Index A_nrows;
  A->nrows(&A_nrows);
  float d = 1.f;

  Vector<float> f1(A_nrows);
  Vector<float> f2(A_nrows);
  std::vector<Index> indices(1, s);
  std::vector<float> values(1, 1.f);
  f1.build(&indices, &values, 1, GrB_NULL);

  v->fill(0.f);
  float c = 1.f;
  while (c > 0) {
    // Assign level d at indices f1 to visited vector v
    graphblas::assign(v, &f1, GrB_NULL, d, GrB_ALL, A_nrows, &desc);
    // Set mask to use structural complement (negation)
    desc->toggle(GrB_MASK);
    // Multiply frontier f1 by transpose of matrix A using visited vector v as mask
    // Semiring: Boolean semiring (see Table 4)
    graphblas::vxm(&f2, v, GrB_NULL, LogicalOrAndSemiring<float>(), &f1, A, &desc);
    // Set mask to not use structural complement (negation)
    desc->toggle(GrB_MASK);
    f2.swap(&f1);
    // Check how many vertices of frontier f1 are active, stop when number reaches 0
    // Monoid: Standard addition (see Table 4)
    graphblas::reduce(&c, GrB_NULL, PlusMonoid<float>(), &f1, &desc);
    d++;
  }
}
```

Algorithm 1. Matrix formulation of BFS (top) and example GraphBLAST code (bottom).



(a) Graph representation

(b) Linear algebraic representation

(c) Graph concepts and linear algebraic equivalents

Fig. 5. Running example of breadth-first-search from source node 1. Currently, we are on level 2 and trying to get to level 3. To do so we need to do a graph traversal from the current frontier (vertices 0, 2, 3) to their neighbors (vertices 2, 4, 5, 7). This corresponds to the multiplication $\mathbf{A}^T\mathbf{f}$. This is followed by filtering out visited vertices (vertices 2, 5), leaving us with the next frontier (vertices 4, 7). This corresponds to the elementwise multiply $\neg\mathbf{v} .* (\mathbf{A}^T\mathbf{f})$.

3.6 Differences with GraphBLAS C API standard

We have tried to make our framework interface adhere to the GraphBLAS C API as close as possible, but since we decided to take advantage of certain C++ features (templates and functors, in particular) in our framework, we have departed from the strict API standard. We observe that some differences are motivation for the design of a GraphBLAS C++ API [14]. Some major differences in the interface are listed below:

(1) We use C++ templates and functors to implement GraphBLAS Semirings. For a more in-depth discussion, see Section 6.2.

(2) We require passing in a template parameter specifying the type in place of: (i) passing a datatype of `GrB_Type` to `Matrix` and `Vector` declaration, (ii) specifying types used in the semiring. This allows more compile-time type-checking to ensure that the types are correct, which is not possible as a compiled binary, but has the disadvantage of the user having to do more work than necessary (i.e., despite the backend knowing what type it is in the `Matrix` declaration).
(3) We provide a header-only C++ library rather than a shared object library. This design choice carries all the advantages and disadvantages header-only libraries have compared to shared object libraries, in addition to the advantages and disadvantages of the first two differences. In our design, this is mainly a consequence of our choice of using C++ templates and functors rather than code generation (see Section 6.2).

A few minor interface differences follow:

(1) We require `Matrix::build` and `Vector::build` to use `std::vector` rather than C-style arrays. However, it would be a simple addition to maintain compatibility with GraphBLAS C API specification by allowing C-style arrays too.
(2) We use a `graphblas` namespace instead of prefixing our methods with `GrB_`.
(3) We provide convenience methods `Vector::fill` and `Descriptor::toggle` that are extensions to the GraphBLAS C API specification.
(4) We choose not to include the `GrB_REPLACE` descriptor setting. This is motivated by our design principle of choosing not to implement what can be composed by a few simpler operations. In this case, if desired, the user can reproduce the `GrB_REPLACE` behavior by first calling `Matrix::clear()` or `Vector::clear()` and then calling the operation they wanted to modify with `GrB_REPLACE`.
(5) We choose to ignore the `accum` input parameter, which is responsible for choosing to accumulate results into the output Vector or Matrix. Our motivation is the same as the above decision. This accumulation can also be done by following up the initial operation by an elementwise addition or elementwise multiply.
(6) We have matrix-vector, matrix-scalar, and vector-scalar variants of elementwise addition and multiplication for convenience and performance. These variants are called rank promotion [57] or Numpy-style broadcasting [43].

4 EXPLOITING INPUT SPARSITY (DIRECTION-OPTIMIZATION)

In this section, we discuss our design philosophy of making exploiting input sparsity and one of its consequences, direction optimization, a first-class feature of our implementation. Since the matrix represents a graph, the matrix $\mathbf{A}$ will be assumed to be stored in sparse format. In traversal-based graph algorithms, the operation we care most about is:

$$\mathbf{y} \leftarrow \mathbf{Ax} \,.* \neg\mathbf{m}$$

Here, the matrix $\mathbf{A}$ represents the graph, the input vector $\mathbf{x}$ represents the current frontier, the output vector $\mathbf{y}$ represents the next iteration frontier, and the mask vector $\mathbf{m}$ represents the set of visited vertices. The negation $\neg$ converts this set of visited vertices to unvisited vertices. For more discussion on masks, see Section 5.

In this section, we try to limit our discussion to *input sparsity*, by which we are referring to the input vector $\mathbf{x}$ being sparse. We exploit this fact to reduce the number of operations. We provide quantitative data to support our conclusion that doing so is of the foremost importance in building a high-performance graph framework. We present three seemingly unrelated challenges with

implementing a linear-algebra-based graph framework based on the GraphBLAS specification, but which we will show are actually facets of the same problem:

(1) Previous work [7, 76] has shown that direction optimization is critical to achieving state-of-the-art performance on breadth-first-search. However, direction optimization has been notably absent in linear-algebra-based graph frameworks and assumed only possible for traditional, vertex-centric graph frameworks. How can direction optimization be implemented as matrix-vector multiplication in a linear-algebra-based framework like GraphBLAS?
(2) The GraphBLAS definition for `mxv` operation is intentionally underspecified. As Figure 4 shows, there are two ways to implement `mxv`. How should it be implemented?
(3) The GraphBLAS definition for `Matrix` and `Vector` objects are intentionally underspecified. What should the underlying data structure for these objects look like?

The results of this section show that the GraphBLAST library would automatically discover the direction optimization idea while only having access to `mxv` function parameters, without having any knowledge of the semantics of the computation (e.g., graph traversals). Because GraphBLAST changes direction based on the sparsity of inputs alone, which are abstract vector and matrix objects, it has the potential to discover more opportunities for push-pull like optimization on domains beyond graph processing.

4.1 Two roads to matrix-vector multiplication

Before we address the above challenges, we draw a distinction between two different ways the matrix-vector multiply $\mathbf{y} \leftarrow \mathbf{Ax}$ can be computed. We distinguish between multiplying a sparse-matrix by a dense-vector (SpMV) and by a sparse-vector (SpMSpV). There is extensive literature focusing on SpMV for GPUs (including a comprehensive survey [35]). However, we concentrate on SpMSpV, because it is more relevant to graph search algorithms where the vector represents the subset of vertices that are currently active and is typically sparse.

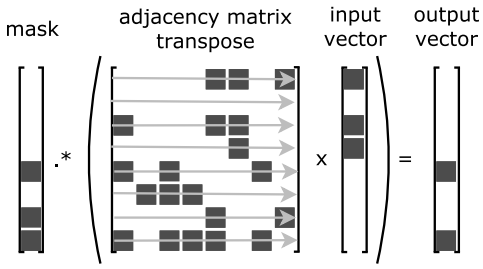
Recall in the running example in Section 3.4 that by exploiting matrix sparsity (SpMV) in favor of dense matrix-vector multiplication (GEMV), we were able to bring the number of load and store instructions down from GEMV's 64 to SpMV's 20. A natural question to ask is whether it is possible to decrease the number of load and store instructions further when the input vector is sparse. Indeed, when we exploit input sparsity (SpMSpV) to get the situation in Figure 6b, we can reduce the number of loads and stores from 20 to 8. Similar to how our move from GEMV to SpMV involved changing the matrix storage format from dense to sparse, moving from SpMV to SpMSpV motivates storing the vector in sparse format. It is worth noting that the sparse vectors are assumed to be implemented as lists of indices and values. A summary is shown in Table 8.

4.2 Related work

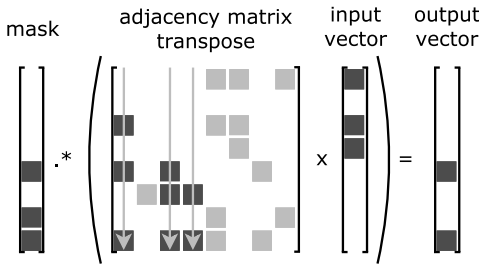
Mirroring the dichotomy between SpMSpV and SpMV, there are two methods to perform one iteration of graph traversal, which are called *push* and *pull*.[2] They can be used to describe graph traversals in a variety of graph traversal-based algorithms such as breadth-first-search, single-source shortest-path, and PageRank.

In the case of breadth-first-search, *push* begins with the current frontier (the set of vertices from which we are traversing the graph) and looks for children of this set of vertices. Then, from this set of children, the previously visited vertices must be filtered out to generate the output frontier (the frontier we use as input on the next iteration). In contrast, *pull* starts from the set of *unvisited* vertices and looks back to find their parents. If a node in the unvisited-vertex set has a parent in the

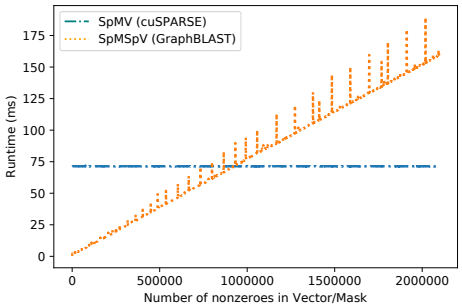
[2]To the authors' best knowledge, the terminology of "push" and "pull" was first introduced by Karp et al. [49] in the context of updates to distributed copies of a database.



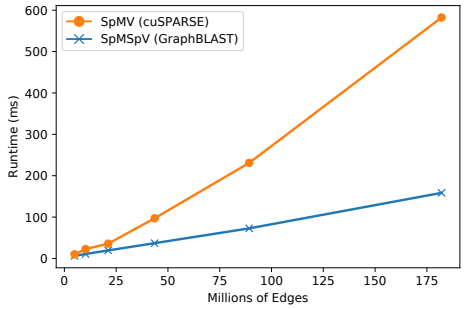
(a) SpMV (20 loads and stores)



(b) SpMSpV (8 loads and stores)



(c) Algorithmic complexity of SpMV and SpMSpV as a function of vector sparsity.



(d) SpMV and SpMSpV runtime on Kronecker scale-16–21 graphs.

Fig. 6. Comparison of SpMV and SpMSpV.

Operation	Mask	Complexity	Matrix Sparsity (**A**)	Input Vector Sparsity (**x**)	Output Vector Sparsity (**m**)
GEMV	no	$O(MN)$			
SpMV (pull)	no	$O(dM)$	✓		
SpMSpV (push)	no	$O(d\ nnz(\mathbf{x}))$	✓	✓	
GEMV	yes	$O(N\ nnz(\mathbf{m}))$			✓
SpMV (pull)	yes	$O(d\ nnz(\mathbf{m}))$	✓		✓
SpMSpV (push)	yes	$O(d\ nnz(\mathbf{x}))$	✓	✓	

Table 8. Computational complexity of matrix-vector multiplication where d is the average number of nonzeroes per row or column, and $\mathbf{A}$ is an M-by-N matrix. The top three rows indicate the standard case $\mathbf{y} \leftarrow \mathbf{Ax}$, while the bottom three rows represent the masked case $\mathbf{y} \leftarrow \mathbf{Ax} .* \mathbf{m}$. Checkmarks indicate which form of sparsity each operation exploits. The notation $nnz(\mathbf{a})$) refers to the number of nonzero entries in a vector $\mathbf{a}$.

current frontier, we add it to the output frontier. Beamer, Asanović, and Patterson [7] observed that in the middle iterations of a BFS on scale-free graphs, the frontier becomes large and each neighbor is found many times, leading to redundant work. They show that for optimal performance, in these intermediate iterations, they should switch to *pull*, and then in later iterations, back to *push*.

Many graph algorithms such as breadth-first-search, single-source shortest-path, and PageRank involve multiple iterations of graph traversals. Switching between *push* and *pull* in different iterations applied to the specific algorithm of breadth-first-search is called *direction optimization* or

direction-optimized BFS, which was also described by Beamer, Asanović, and Patterson [7]. This approach is also termed *push-pull*. Building on this work, Shun and Blelloch [76] generalized direction optimization to graph traversal algorithms beyond BFS. To avoid confusion with the BFS-specific instance, we refer to Shun and Blelloch's contribution as *generalized direction optimization*.

Beamer, Asanović and Patterson later studied matrix-vector multiplication in the context of SpMV- and SpMSpV-based implementations for PageRank [8]. In both their work and that of Besta et al. [11], the authors noted that switching between push/pull is the same as switching between SpMSpV/SpMV. In both works, authors show a one-to-one correspondence between push and SpMSpV, and between pull and SpMV; they are two ways of thinking about the same concept.

Our work differs from Beamer, Asanović, and Patterson and Besta et al. in three ways: (1) they emphasize graph algorithm research, whereas we focus on building a graph framework, (2) their work targets multithreaded CPUs, while ours targets GPUs, and (3) their interface is vertex-centric, but ours is linear-algebra-based.

The work we present here builds on our earlier work and is first to extend the *generalized direction optimization* technique to linear-algebra-based frameworks based on the GraphBLAS specification. In contrast, previous implementations to the GraphBLAS specification, such as GBTL [88] and SuiteSparse [27], do not support *generalized direction optimization* and as a consequence, trail state-of-art graph frameworks in performance.

In both implementations, the operation `mxv` is implemented as a special case of `mxm` when one of the matrix dimensions is 1 (i.e., is a `Vector`). The `mxm` implementation is a variant of Gustavson's algorithm [41], which takes advantage of both matrix sparsity and input vector sparsity, so it has a similar performance characteristic as SpMSpV. Therefore, it shares SpMSpV's poor performance when either: (1) there are more elements in the input vector or (2) when there are fewer elements in the mask (representing fewer operations that need to be performed). In other words, neither GBTL and SuiteSparse automatically switch to *pull* when the input vector becomes large in the middle iterations of graph traversal algorithms like BFS, and perform *push* throughout the entire BFS. In comparison, our graph framework balances exploiting input vector sparsity (SpMSpV) with the efficiency of SpMV during iterations of high input vector sparsity. This helps us match or exceed the performance of existing graph frameworks (Section 8).

4.3 Implementation

In this subsection, we revisit the three challenges we claimed boil down to different facets of the same challenge: exploiting input sparsity.

Direction optimization Our backend automatically handles direction optimization when `mxv` is called, by consulting an empirical cost model and calling either the SpMV or SpMSpV routine that we expect will result in the fewest memory accesses.

mxv: SpMV or SpMSpV Both routines are necessary for an efficient implementation of `mxv` in a graph framework.

Matrix and Vector storage format For `Matrix`, store both CSR and CSC, but give users the option to save memory by only storing one of these two representations. The result is a memory-efficient, performance-inefficient solution. For `Vector`, since both dense vector and sparse vector are required for the two different routines SpMV and SpMSpV respectively, we give the backend the responsibility to switch between dense and sparse vector representations. We allow the user to specify the initial storage format of the `Matrix` and `Vector` objects.

4.3.1 Direction optimization. When a user calls `mxv`, our backend chooses either the SpMV or SpMSpV routine, using an empirical cost model to select the one with fewer memory accesses.

Work	Direction	Criteria	Application
Beamer et al. [7]	push → pull	$\|E_f\| > \|E_u\|/14$ and increasing	BFS only
	push ← pull	$\|V_f\| < \|V\|/24$ and decreasing	BFS only
Ligra [76]	push → pull	$\|E_f\| > \|E\|/20$	generalized
	push ← pull	$\|E_f\| < \|E\|/20$	generalized
Gunrock [82]	push → pull	$\|E_f^*\| > \|E_u^*\|/1000$	BFS only
	push ← pull	$\|E_f^*\| < \|E_u^*\|/5$	BFS only
This work	push → pull	$\|E_f^*\| > \|E\|/10$	generalized
	push ← pull	$\|E_f^*\| < \|E\|/10$	generalized

Table 9. Direction-optimization criteria for four different works. $|V_f|$ indicates the number of nonzeroes in the frontier $\mathbf{f}$. $|E_f|$ indicates the number of neighbors from the frontier $\mathbf{f}$. $|E_u|$ indicates the number of neighbors from unvisited vertices. Superscript * indicates the value is approximated rather than precisely calculated.

Table 9 shows how our decision to change directions compares with existing literature. We make the following simplifying assumptions:

(1) On GPUs, computing the precise number of neighbors $|E_f|$ for a given frontier $\mathbf{f}$ requires prefix-sum computations. To avoid what Beamer et al. called a non-significant amount of overhead, we instead approximate the precise number of neighbors using the number of nonzeroes in the vector by assuming that in expectation, each element in the vector has around the same number of neighbors, i.e., $d|V_f| \approx |E_f|$. Gunrock also makes this assumption.
(2) When the mask (representing the unvisited vertices) is dense, counting the number of unvisited vertices $|V_u|$ requires an additional GPU kernel launch, which represents significant overhead (Section 2.1.4). Therefore, we make the assumption that the number of unvisited vertices is all vertices, i.e., $|V_u| \approx |V|$ so $|E_u| \approx |E|$. We find this is a reasonable assumption to make, because for scale-free graphs the optimal time to switch from push to pull is very early on, so $|V_u| \approx |V|$. Ligra also makes this assumption.

4.3.2 mxv: SpMV or SpMSpV. Following our earlier work [86], which showed that SpMV is not performant enough for graph traversal and that SpMSpV is necessary, we run our own microbenchmark regarding GraphBLAS. In our microbenchmark, we benchmarked `graphblas::mxv` implemented with two variants—`SparseVector` and `DenseVector`—as a function of `Vector` sparsity for a synthetic undirected Kronecker graph with 2M vertices and 182M edges. For more details of this experiment, see Section 8.

As our microbenchmark in Figure 6 illustrates, the performance of the SpMSpV variant of `graphblas::mxv` is proportional to the sparsity of the input vector. However, the SpMV variant is constant. This matches the theoretical complexity shown in Table 8, which shows that SpMV scales with $O(dM)$, which is independent of input vector sparsity. However, SpMSpV is able to factor in the sparsity of the input vector (i.e., $nnz(\mathbf{x})$) into the computational cost. For more details, see Section 6.3.1.

4.3.3 Matrix and Vector storage format. One of the most important design choices for an implementer is whether `Matrix` and `Vector` objects ought to be stored in dense or sparse storage, and if sparse, which type of sparse matrix or vector storage?

For `Matrix` objects, the decision is clear-cut. Since graphs tend to have more than 99.9% sparsity and upwards of millions of vertices, storing them in dense format would be wasteful and in some cases impossible because of the limitation of available device memory. We use the popular CSR

(Compressed Sparse Row) format, because it is standard in graph analytics and supports the fast row access required by SpMV. Similarly, since we also need to support SpMSpV and fast column access, we also support the CSC data structure (Figure 6).

For `Vector` objects, we support both dense and sparse storage formats. The dense storage is a flat array of values. The sparse storage is a list of sorted indices and values for all nonzero elements in the vector. Through additional `Vector` object methods `Vector::buildDense` and `Vector::fill` (shown in Table 4), we allow users to give the backend hints on whether they want the object to initially be stored in dense or sparse storage.

4.4 Direction optimization insights

Exploiting input sparsity is a useful and important strategy for high-performance in graph traversals. We believe that the GraphBLAS interface decision where users do not have to specify whether or not they want to exploit input sparsity is a good one; we showed that instead, users must only write code once using the `mxv` interface and both forms of SpMV and SpMSpV code can be automatically generated for them by GraphBLAST. In the next section, we will show how the number of memory accesses can also be reduced by exploiting output sparsity.

5 EXPLOITING OUTPUT SPARSITY (MASKING)

The previous section discussed the importance of reducing the number of load and store instructions using input vector sparsity. This section deals with the mirror situation, which is *output vector sparsity* (or *output sparsity*). Output vector sparsity can also be referred to as an output mask or *masking* for short.

Masking allows GraphBLAS users to tell the framework they are planning to follow a matrix-vector or matrix-matrix multiply with an elementwise product. This allows the backend to implement the fused mask optimization, which in some cases may reduce the number of computations needed. Alongside exploiting input sparsity, our design philosophy was to make exploiting output sparsity a first-class feature in GraphBLAST with highly efficient implementations of masking. Masking raises the following implementation challenges.

(1) Masking is a novel concept introduced by the GraphBLAS API to allow users to decide which output indices they do and do not care about computing. How can masking be implemented efficiently?
(2) When should the mask be accessed before the computation in out-of-order fashion and when should it be processed after the computation?

5.1 Motivation and applications of masking

Following the brief introduction to masking in Section 5, the reader may wonder why such an operation is necessary. Masking can be thought of in two ways: (i) masking is a way to fuse an element-wise operation with another operation from Table 7; and (ii) masking allows the user to express for which indices they do and do not require a value before the actual computation is performed. We define this as *output sparsity*. The former means that masking is a way for the user to tell the framework there is an opportunity for kernel fusion, while the latter is an intuitive way to understand why masking can reduce the number of computations in graph algorithms.

There are several graph algorithms where exploiting *output sparsity* can be used to reduce the number of computations:

(1) In breadth-first-search [17, 85], the mask `Vector` represents the *visited* set of vertices. Since in a breadth-first-search each vertex only needs to be visited once, the user can let the software know that the output need not include any vertices from the *visited* set.

(2) In single-source shortest-path [26], the mask `Vector` represents the set of vertices that have seen their distances from the source vertex change in this iteration. The mask can thus be used to zero out currently active vertices from the next traversal, because their distance information has already been taken into account in earlier traversal iterations. The mask can be used to help keep the active vertices `Vector` sparse throughout the SSSP; otherwise, it would be increasingly densifying.
(3) In adaptive PageRank (also known as PageRankDelta) [48, 76], the mask `Vector` represents the set of vertices that has converged already. The PageRank value for this set of vertices does not need to be updated in future iterations.
(4) In triangle counting [2, 83], the mask `Matrix` represents the adjacency matrix where a value 1 at $\mathbf{M}(i, j)$ indicates the presence of edge $i \rightarrow j$, and 0 indicates a lack of an edge. Performing a dot product $\mathbf{MM}$ corresponds to finding for each index pair (i, j) the number of wedges $i \rightarrow k \rightarrow j$ that can be formed for all $k \in V$. Thus applying the mask `Matrix` to the dot product will yield $\mathbf{MM} .* \mathbf{M}$, which indicates the set of wedges that are also triangles by virtue of the presence of edge $i \rightarrow j$. Here the $.*$ operation indicates element-wise operation. To get the number of wedges from the set of wedges, a further reduction is required. The algorithm described was to explain the purpose of the mask in triangle counting rather than for giving an optimal algorithm. For a better algorithm that does 6× less work, see Section 7.5.

5.2 Microbenchmarks

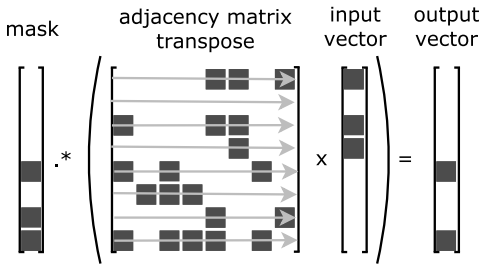
Similar to our earlier microbenchmark (Section 4.3.2), we benchmark how using masked SpMV and SpMSpV variants of `graphblas::mxv` performed compared with unmasked SpMV and SpMSpV as a function of mask `Vector` sparsity for a synthetic undirected Kronecker graph with 2M vertices and 182M edges. For more details of the experiment setup, see Section 8.

As our microbenchmark in Figure 7 illustrates, the masked SpMV variant of `graphblas::mxv` scales with the sparsity of the mask `Vector`. However, the masked SpMSpV is unchanged from the unmasked SpMSpV. This too matches the theoretical complexity shown in Table 8, which shows that masked SpMV scales with $O(d\ nnz(\mathbf{m}))$, where $\mathbf{m}$ is the mask `Vector`. However in our implementation, masked SpMSpV only performs the elementwise multiply with the mask after the SpMSpV operation, so it is unable to benefit from the mask's sparsity. After the columns of the sparse matrix are loaded, it may be possible to use binary search into the mask vector to reduce the number of operations required for the multiway merge, but this still does not asymptotically reduce the number of load and store instructions into the sparse matrix as in the masked SpMV case.

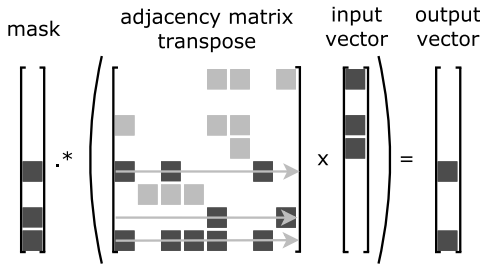
In the running example, recall in Figure 7a that standard SpMV, which performs the matrix-vector multiply followed by the elementwise product with the mask, took 20 load and store instructions. However, when we reverse the sequence of operations by first loading the mask, seeing which elements of the mask are nonzero, and then only doing the matrix-vector multiply for those rows that map to a nonzero mask element, we see that the number of loads and stores drops significantly from 20 down to 10.

5.3 Masking insights

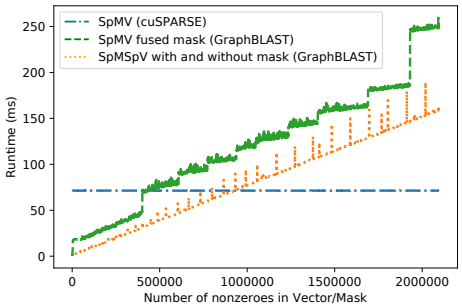
One simple implementation of masking is to perform the matrix multiplication, and then apply the mask to the output. This approach has the benefit of being straightforward and easy to implement. However, we identify two scenarios in which accessing the mask ahead of the matrix multiplication is beneficial:



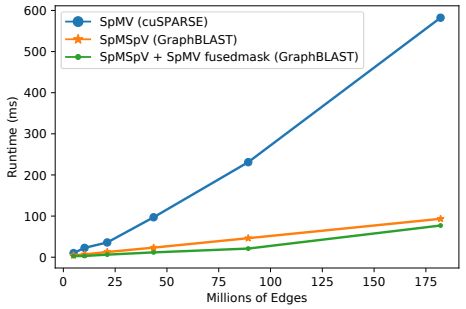
(a) SpMV not fused with mask (20 loads and stores).



(b) SpMV fused with mask (10 loads and stores).



(c) Algorithmic complexity as a function of vector and mask sparsity: (i) SpMV without mask, (ii) SpMV with mask, (iii) SpMSpV without mask, (iv) SpMSpV with mask. Note: Since SpMSpV with and without mask produced the same runtime, we lump them as "SpMSpV with and without mask".



(d) SpMV and SpMSpV runtime on Kronecker scale-{16–21} graphs.

Fig. 7. Comparison with and without fused mask.

	mxm first		mask first			
Dataset	Nonzeroes	Runtime (s)	Nonzeroes	Runtime (s)	Memory savings	Speedup
coAuthorsCiteseer	2.03M	458.3	814K	5.96	2.49×	76.9×
coPapersDBLP	81.3M	3869	15.2M	78.66	5.35×	13.2×
road_central	29.0M	3254	16.9M	246.4	1.72×	49.2×

Table 10. Runtime in milliseconds and speedup of accessing mask before `mxm` and after `mxm` on three datasets. Nonzeroes means how many nonzero elements are in the output of the `mxm`.

(1) Masked `mxv`: As Figure 7 illustrates, the masked SpMV is advantageous and to be preferred when the input vector nonzero count surpasses some threshold. Table 9 is a good starting point at finding the optimal threshold for given hardware.
(2) Masked `mxm`: Table 10 shows two benefits of accessing the mask before doing the `mxm`. The first benefit is lower memory consumption. Typically, `mxm` generates an order of magnitude more nonzeroes in the output matrix compared with the two input matrices, which in the absence of kernel fusion must be saved and typically causes out-of-memory errors. By accessing the

mask first, this order of magnitude blow-up in nonzeroes can be avoided. Using the mask as an oracle, the mask yields an upper bound in where nonzeroes can be generated. Therefore, an order of magnitude less computation can be done by accessing the mask to determine nonzeroes i, j s.t. $\mathbf{M}(i, j) \neq 0$, then loading only $\mathbf{A}(i, :)$ and $\mathbf{B}(:, j)$, performing the dot product between the two, and writing the result to $\mathbf{C}(i, j)$. Therefore, the second benefit is from avoiding computation.

6 GPU IMPLEMENTATION

In this section, we discuss implementation details specific to the GPU.

6.1 Memory management

GBTL uses the Thrust template library to provide wrappers around GPU memory that automatically handle CPU-to-GPU and GPU-to-CPU communication. We instead decided to manually manage GPU pointers ourselves. This offers graph algorithm developers (i.e., the users) the same seamless experience of not having to concern themselves with whether the data is on the CPU or the GPU.

For the concrete implementation of each object—SparseVector, DenseVector, SparseMatrix, DenseMatrix, etc.—we keep both a canonical GPU copy and a CPU copy that is allowed to go out-of-date. Upon initialization, we maintain the canonical copy on the GPU at all times and use a flag to keep track whether or not the CPU version differs from the canonical GPU copy and is stale. When operations mutating the GPU copy are run, this flag will get set to `true` indicating the CPU copy is now stale. For operations that interact with the world external to GraphBLAS such as `extract` and `extractTuples`, we will copy data back to the CPU depending on whether the flag tells us the CPU copy is stale or not. If it is not stale, we can return the CPU copy directly. If it is stale, we will copy data back to the CPU and reset the flag `false` indicating the GPU and CPU copy have equal values.

On GPUs, memory allocation time can be a significant fraction of runtime. Since some operations require temporary memory, we keep a memory pool of already-allocated GPU memory. Currently, we associate this memory pool with the `Descriptor` object, because we do not find we often require more than one `Descriptor`. In the future, we will consider changing this to use the Factory pattern [52], which is considered standard design for memory pools.

6.2 Operators

One of the biggest challenges of implementing GraphBLAS is solving the problem of supporting the large cross product of possible functionalities. Consider, for example, `Semiring`. This operator needs to support 11 built-in GraphBLAS types (and any user-defined type), 22 built-in binary operations, and 8 built-in monoid operations. Even without user-defined types, this comes to a total of 1459 operators for the 3 GraphBLAS methods that take a `Semiring`, `mxv`, `vxm`, and `mxm`.

In the literature [14, 27, 88], two ways have been proposed to tackle this problem:

(1) Using code generation tools and macros.
(2) Using C++ templates and functors. This is the approach taken by GBTL, which we adopt here as well.

The first method is the approach taken by SuiteSparse [27]. This has the advantage of being a shared object library, which can be linked to using frontends written in interpreted languages. We instead adopt the second method, which has the advantage of not needing to maintain code generation tools and macros, which can be challenging.

To express monoids and semirings, we use `__host__` and `__device__` functors that overload the function call operator (i.e. `operator()`). We use a macro that constructs structs composed of

one and two of these functors for monoids and semirings respectively. The macro also takes an identity-element input.

6.3 Implementation of key primitives and load balancing

What follows are the implementation details and load balancing techniques behind the four key primitives SpMSpV, SpMV, SpMM and masked SpGEMM. Load balancing attempts to distribute work equally across the GPU's processors (threads, warps and blocks). To motivate the need for load balance, consider an implementation that assigns each matrix row to a different processor. Because the number of nonzeroes per row may vary greatly, the amount of work per processor may also vary greatly, leading to inefficient execution. We use the following strategies to implement load-balanced kernels:

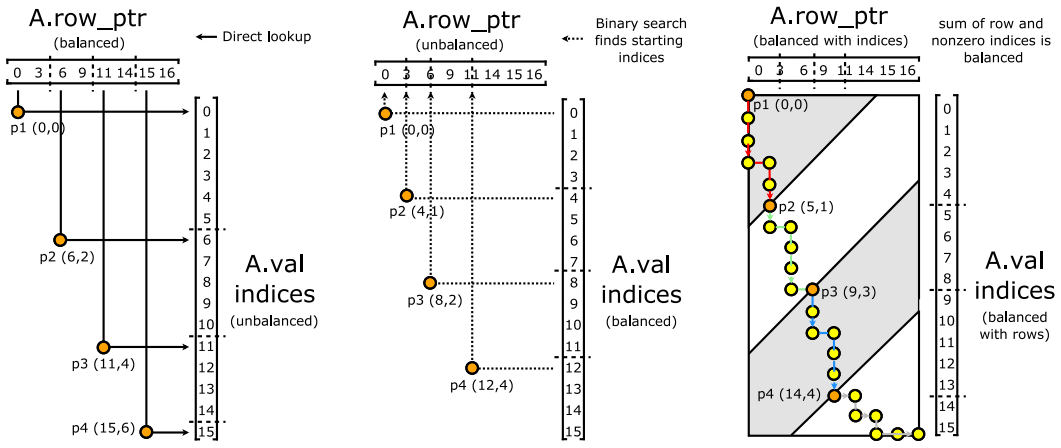
(1) Multiple kernel approach: SpMSpV
(2) Merge-based: SpMV
(3) Merge-based and Row split: SpMM
(4) Row split: masked SpGEMM

6.3.1 SpMSpV. Based on our earlier work [85, 86], our current SpMSpV implementation is composed of several steps. It heavily relies on scan primitives and in particular the `IntervalExpand` and `IntervalGather` operations of ModernGPU [4]. Recall that in SpMSpV, each column i is multiplied by $\mathbf{x}(i)$ if and only if $\mathbf{x}(i) \neq 0$. This operation costs $\text{flops}(\mathbf{A}, \mathbf{x}) = \sum_{i|\mathbf{x}(i)\neq 0} nnz(\mathbf{A}(:, i))$. `IntervalExpand` creates a temporary vector of length $\text{flops}(\mathbf{A}, \mathbf{x})$. This allows data-parallel multiplication of all the nonzeros that will contribute to the final output with their corresponding vector values. Note that the extra memory consumption for this temporary workspace is smaller than the input matrix size (unless the vector is completely dense, in which case we would be calling SpMV instead of SpMSpV). These intermediate values are then sorted using RadixSort in linear time to bring identical indices next to each other. Finally, a segmented reduction (ModernGPU's `ReduceByKey`) creates the desired output vector. Thus, SpMSpV has $O(\text{flops}(\mathbf{A}, \mathbf{x}))$ work and $O(\lg(\text{flops}(\mathbf{A}, \mathbf{x})))$ depth.

6.3.2 SpMV. For SpMV, we use the nonzero-split implementation of the merge-based algorithm provided by ModernGPU [4]. In benchmarking, we find that performance is similar to the merge path algorithm by Merrill and Garland [59]. The difference between the two algorithms is shown in Figure 8 and may be described as follows:

(1) Row split [9]: Assigns an equal number of rows to each processor.
(2) Merge-based: Performs two-phase decomposition—the first kernel divides work evenly amongst CTAs, then the second kernel processes the work.
   (a) Nonzero-split [4, 25]: Assign an equal number of nonzeros per processor. Then do a 1-D (1-dimensional) binary search on *row offsets* to determine at which row to start.
   (b) Merge path [59]: Assign an equal number of {nonzeros and rows} per processor. This is done by doing a 2-D binary search (i.e., on the diagonal line in Figure 8c) over *row offsets* and *nonzero indices* of matrix $\mathbf{A}$.

The merge path algorithm has the advantage of doing well in the pathological case of arbitrarily many empty matrix rows, which can cause an arbitrarily large amount of load imbalance for ModernGPU's nonzero-split. However, we find that this does not happen often in practice. Hence, GraphBLAST currently uses ModernGPU's SpMV implementation that relies on the segmented-scan primitive. In theory, segmented-scan-based SpMV has depth logarithmic on the number of nonzeros involved [13], and it can be implemented on a GPU in a work-efficient way [72]. While in



(a) Row split (b) Merge-based (nonzero split) (c) Merge-based (merge path)

Fig. 8. The three parallelizations for CSR SpMV and SpMM on matrix **A**. The orange markers indicate the segment start for each processor ($P = 4$).

practice one has to choose a block size on the GPU, Sengupta et al. [71] showed that this does not increase work and depth of the segmented scan implementation asymptotically.

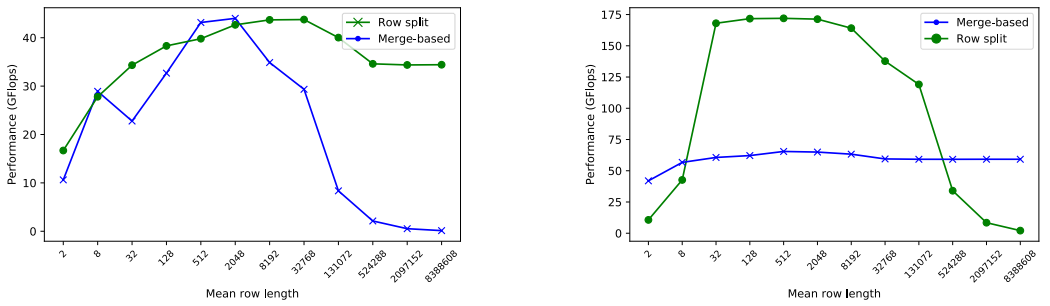
We show the results of a microbenchmark comparing the merge-based algorithm against cuSPARSE's implementation of SpMV in Figure 9a, which we suspect relies on the row split algorithm. The experimental setup is described in Section 8. The right side of the x-axis represents load imbalance where long matrix rows are not divided enough, resulting in some computation resources on the GPU remaining idle while others are overburdened. The left size of the x-axis represents load imbalance where too many computational resources are allocated to each row, so some remain idle. From this figure, it is clear that the merge-based algorithm is superior to the row split algorithm at addressing these two types of load imbalance, despite there being two configurations 512 and 2048 for which row split is faster.

6.3.3 SpMM. For SpMM, we have in our previous work [84] extended the approach taken by SpMV. As Figure 9b shows, our conclusions from earlier work show that while merge-based SpMM does help with solving the two types of load balance as in SpMV, we have identified problems scaling the algorithm when there are many columns in the right-hand-side matrix. Therefore, based on this benchmark and experimentation using 157 SuiteSparse matrices, we developed a multi-algorithm:

(1) When $nnz/M < 9.35$: Use merge-based algorithm.
(2) When $nnz/M \geq 9.35$: Use row split algorithm.

The reason that this heuristic does not capture the right side of Figure 9b, where merge-based is superior, is that in practice we did not encounter any matrices that had a mean row length of greater than 524,288.

6.3.4 Masked SpGEMM. In our implementation, we use a generalization of this primitive where we assume we are solving the problem for three distinct matrices $\mathbf{C} = \mathbf{AB} .* \mathbf{M}$. We use a straightforward row split where we assign a warp per row of the mask **M**, and for every nonzero $\mathbf{M}(i, j)$ in the mask, each warp loads the row $\mathbf{A}(i, :)$ in order to perform the dot product $\mathbf{A}(i, :)\mathbf{B}(:, j)$. Using their **A**-elements, each thread in the warp performs binary search on column $\mathbf{B}(:, j)$ and accumulates



(a) SpMV (b) SpMM

Fig. 9. Microbenchmark showing performance of the merge-based algorithm compared against the row-split based algorithm for SpMV and SpMM.

the result of the multiplication. After the row is finished, we perform a warp reduction and write the output to $\mathbf{C}(i, j)$.

We also experimented with first computing the SpGEMM and then applying the mask. However, as Table 10 shows, our direct Masked SpGEMM implementation was 13–79× faster and used significantly less memory.

7 GRAPH ALGORITHMS

One of the main advantages of GraphBLAS is that its operations can be composed to develop new graph algorithms. For each graph algorithm in this section, we describe the hardwired GPU implementation of that algorithm and how our implementation can be expressed using GraphBLAS. Then the next section will compare performance between hardwired and GraphBLAS implementations. Figure 10 shows the GraphBLAS algorithms required to implement each algorithm.

We chose the five graph algorithms BFS, SSSP, PR, CC, and TC. Based on Beamer's thorough survey of graph processing frameworks in his Ph.D. dissertation [6], they represent all five of the most commonly evaluated graph algorithms. In addition, they stress different components of graph frameworks. BFS stresses the importance of masking and being able to quickly filter out nonzeros that don't have an associated value. SSSP stresses masking and being able to run SpMV on nonzeros with an associated value representing distance. PR stresses having a well-load-balanced SpMV. CC tests expressibility and random memory accesses from hooking and pointer-jumping. TC stresses having a masked SpGEMM implementation.

7.1 Breadth-first-search

Given a source vertex $s \in V$, a BFS is a full exploration of a graph G that produces a spanning tree of the graph, containing all the edges that can be reached from s, and the shortest path from s to each one of them. We define the depth of a vertex as the number of hops it takes to reach this vertex from the root in the spanning tree. The visit proceeds in steps, examining one BFS level at a time. It uses three sets of vertices to keep track of the state of the visit: the *frontier* contains the vertices that are being explored at the current depth, *next* has the vertices that can be reached from *frontier*, and *visited* has the vertices reached so far. BFS is one of the most fundamental graph algorithms and serves as the basis of several other graph algorithms.

Hardwired GPU implementation The best-known BFS implementation of Merrill et al. [60] achieves its high performance through careful load-balancing, avoidance of atomics, and

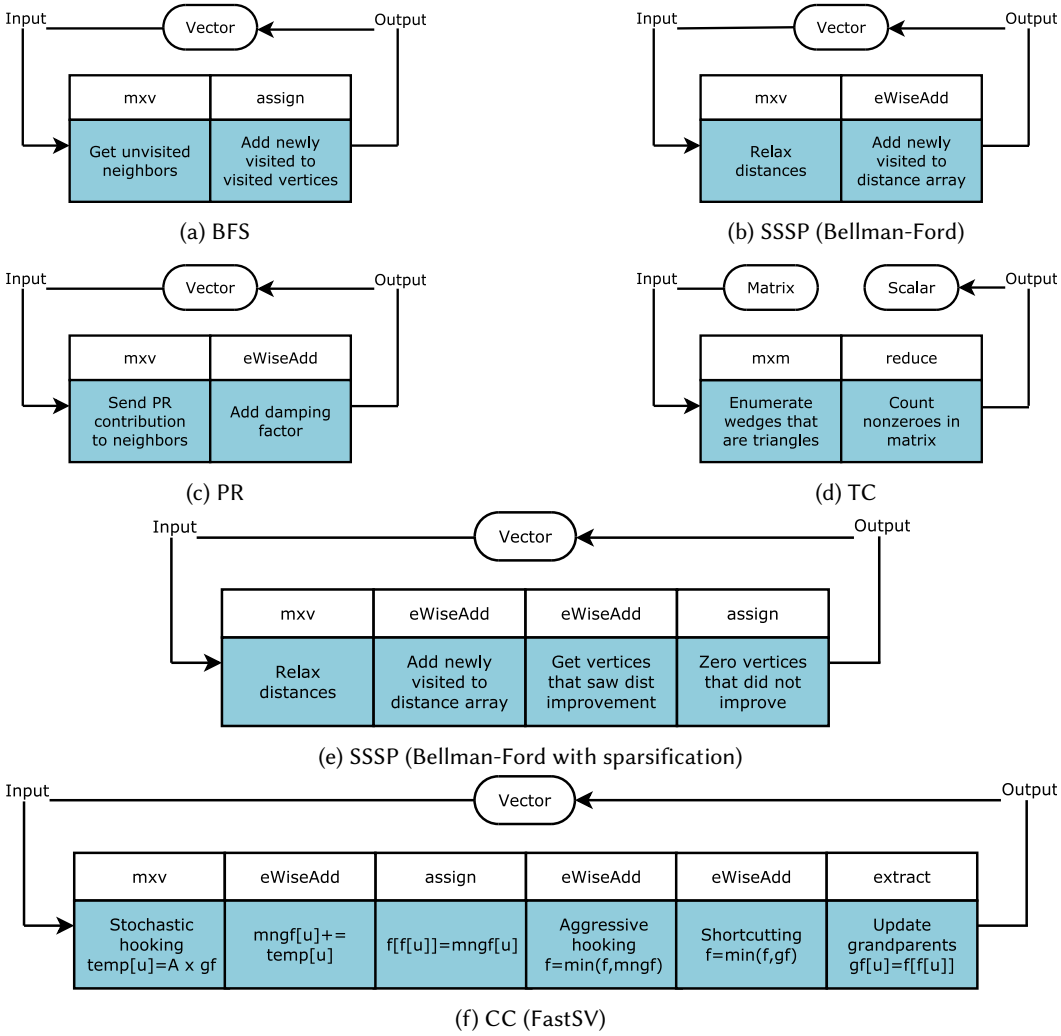


Fig. 10. Operation flowchart for different algorithms expressed in GraphBLAS. A loop indicates a while-loop that runs until the Vector is empty.

heuristics for avoiding redundant vertex discovery. Its chief operations are expand (to generate a new frontier) and contract (to remove redundant vertices) phases. Enterprise [55], a GPU-based BFS system, introduces a very efficient implementation that combines the benefits of the direction optimization of Beamer, Asanović and Patterson [7], leverages the adaptive load-balancing workload mapping strategy of Merrill et al., and chooses to not synchronize each BFS iteration, which addresses the kernel launch overhead problem (Section 2.1.4).

GraphBLAST implementation Merrill et al.'s expand and contract maps nicely to GraphBLAST's `mxv` operator with a mask using a Boolean semiring. Like Enterprise, we implement efficient load-balancing (Section 6) and direction optimization, which was described in greater detail in Section 4. We do not use Enterprise's method of skipping synchronization between BFS iterations, but we use two optimizations: *early-exit* and *structure-only*, which are consequences

of the Boolean semiring that is associated with BFS. We also use *operand reuse*, which avoids having to convert from sparse to dense during direction optimization. These optimizations are inspired by Gunrock and are described in detail by the authors in an earlier work [85]. If our implementation were to always choose the top-down direction, it would have work complexity of $O(nnz(\mathbf{A})) = O(|E|)$ and depth $O(D \cdot \log d_{\max})$ where D is the graph diameter and $d_{\max}$ is the maximum vertex degree. This bound follows from the SpMSpV complexity described in Section 6.3.1. We are not aware of a worst-case analysis of the direction-optimized search but most switching heuristics are conservative and only switch to the bottom-up direction when it decreases work. Given that analysis in Section 6.3.2 shows that masked-SpMV depth is no larger than SpMSpV depth, we conclude that our implementation has worst-case work $O(nnz(\mathbf{A})) = O(|E|))$ and depth $O(D \cdot \log d_{\max})$.

7.2 Single-source shortest-path

Given a source vertex $s \in V$, a SSSP is a full exploration of weighted graph G that produces a distance array of all vertices v reachable from s, representing paths from s to each v such that the path distances are minimized.

Hardwired GPU implementation Currently the highest-performing SSSP algorithm implementation on the GPU is the work from Davidson et al. [26]. They provide two key optimizations in their SSSP implementation: (1) a load balanced graph traversal method, and (2) a priority queue implementation that reorganizes the workload.

GraphBLAST implementation We take a different approach from Davidson et al. to solve SSSP. We show that our approach both avoids the need for ad hoc data structures such as priority queues and wins in performance. The optimizations we use are: (1) *generalized direction optimization*, which is handled automatically within the `mxv` operation rather than inside the user's graph algorithm code, and (2) sparsifying the set of active vertices after each iteration by comparing each active vertex to see whether or not it improved over the stored distance in the distance array. The second phase introduces two additional steps (compare Figures 10b and 10e). These facts make our SSSP implementation an adaptive variant of Bellman-Ford. While the worst-case work complexity is $O(|E|\,|V|))$, it is much faster in practice due to (a) convergence being achieved significantly before $|V|$ iterations, and (b) direction optimization. Each iteration has only $O(\log d_{\max})$ depth.

7.3 PageRank

The PageRank link analysis algorithm assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of quantifying its relative importance within the set. The iterative method of computing PageRank gives each vertex an initial PageRank value and updates it based on the PageRank of its neighbors (this is the "pull" formulation of PageRank), until the PageRank value for each vertex converges. There are variants of the PageRank algorithm that stop computing PageRank for vertices that have converged already and also remove it from the set of active vertices. This is called adaptive PageRank [48] (also known as PageRankDelta). In this paper, we do not implement or compare against this variant of PageRank. We also acknowledge that different kinds of iterative solvers can be used for computing PageRank [39].

Hardwired GPU implementation One of the highest-performing implementations of PageRank is written by Khorasani, Vora, and Gupta [51]. In their system, they use solve the load imbalance and GPU underutilization problem with a GPU adoption of GraphChi's Parallel Sliding Window scheme [53]. They call this preprocessing step "G-Shard" and combine it with

a concatenated window method to group edges from the same source IDs. We realize that due to G-Shard's preprocessing this comparison is not exactly fair to GraphBLAS, but include the comparison, because they are one of the leaders in PageRank performance and despite their preprocessing, our dynamic load-balancing is sufficient to make our implementation faster in the geomean (geometric mean).

Gunrock implementation Gunrock supports both pull- and push-based PageRank; its push-based implementation is in general faster than its pull-based implementation. For a fair algorithmic comparison, we measure against Gunrock's pull-based implementation.

GraphBLAST implementation In GraphBLAST, we rely on the merge-based load-balancing scheme discussed in Section 6. The advantage of the merge-based scheme is that unlike Khorasani, Vora, and Gupta, we do not need any specialized storage format; the GPU is efficient enough to do the load-balancing on the fly. In terms of exploiting input sparsity, we demonstrate that our system is intelligent enough to determine that we are doing repeated matrix-vector multiplication and because the vector does not get any sparser, it is more efficient to use SpMV rather than SpMSpV. As described in Section 6.3.2, our SpMV implementation is based on ModernGPU [4], which uses the segmented-scan primitive with logarithmic depth and linear work. Consequently, an iteration of PageRank has $O(\tilde{e})$ work and $O(\lg \tilde{e})$ depth, where $\tilde{e} = \sum_{i|\mathbf{m}(i)\neq 0} nnz(\mathbf{A}(i,:))$ is the number of nonzeros that need to be touched in that iteration. Note that $\tilde{e}$ is at most $nnz(\mathbf{A})$ but is often smaller due to already converged vertices.

7.4 Connected components

Weakly connected components (abbreviated as connected components) is the problem of: (1) identifying all subgraphs (or components) in an undirected graph such that every pair of vertices in the subgraph are connected by edges and no edges connect vertices in different subgraphs, and (2) labeling each vertex with its component ID.

Hardwired GPU implementation Soman, Kishore and Narayanan [79] base their GPU implementation on two algorithms from the PRAM literature: hooking and pointer-jumping. Hooking takes an edge as the input and tries to set the component IDs of the two end vertices of that edge to the same value. In odd-numbered iterations, the lower vertex writes its value to the higher vertex, and vice versa in the even-numbered iteration. This strategy is used to increase the rate of convergence over a more naive approach such as a breadth-first-search. Pointer-jumping reduces a multi-level tree in the graph to a one-level tree (star). By repeating these two operators until no component ID changes for any node in the graph, the algorithm will compute the number of connected components for the graph and the connected component to which each node belongs.

Gunrock implementation Gunrock uses a filter operator on an edge frontier to implement hooking. The frontier starts with all edges and during each iteration, one end vertex of each edge in the frontier tries to assign its component ID to the other vertex, and the filter step removes the edge whose two end vertices have the same component ID. We repeat hooking until no vertex's component ID changes and then proceed to pointer-jumping, where a filter operator on vertices assigns the component ID of each vertex to its parent's component ID until it reaches the root. Then a filter step removes the node whose component ID equals its own node ID. The pointer-jumping phase also ends when no vertex's component ID changes.

GraphBLAST implementation GraphBLAST's implementation of CC is based on the FastSV algorithm [90]. FastSV is a linear-algebraic connected components algorithm [89] that is based

on the classic PRAM algorithm of Shiloach and Vishkin [75] that uses hooking and pointer-jumping. We make two interesting observations. (1) We include a *sparsification* optimization that is discussed in the FastSV paper [90]. With 3 lines of code that zero out the redundant values, our push-pull design is able to handle this optimization automatically. (2) We avoid unnecessary GPU-to-CPU and CPU-to-GPU memory copies that would otherwise be required in the original FastSV GraphBLAS implementation using 2 new variants of `assign` and `extract`. Instead of using the `Index *` found with the standard variants, we use a GraphBLAS `Vector` that will implicitly have its values treated as the indices. Since `Vector` will always have the most recent data in GPU memory, this solves the problem of not being able to deduce whether the `Index` pointer is located in CPU or GPU memory. Unlike the other linear-algebraic CC implementation (LACC), FastSV does not guarantee $O(\log |V|)$ worst-case iteration bound because it avoids unconditional hooking for higher performance [89]. We chose FastSV to include in GraphBLAST because it is consistently faster than LACC in practice, despite having worse complexity bounds. Consequently, our CC implementation also takes worst-case $O(|E|\,|V|)$ time and $O(|V|)$ depth. However, it is much faster in practice, similar to other quadratic time and linear depth CC algorithms such as multistep-CC [78].

7.5 Triangle counting

Triangle counting is the problem of counting the number of unique triplets u, v, w in an undirected graph such that $\{(u, v), (u, w), (v, w)\} \in E$. Many important measures of a graph are triangle-based, such as clustering coefficient and transitivity ratio.

Hardwired GPU implementation One of the best-performing implementation of triangle counting is by Bisson and Fatica [12]. In their work, they demonstrate an effective use of a static workload mapping of thread, warp, block per matrix row together with using bitmaps.

GraphBLAST implementation In GraphBLAST, we follow Azad and Buluç [2] and Wolf et al. [83] in modeling the TC problem as a masked matrix-matrix multiplication problem. Given an adjacency matrix of an undirected graph $\mathbf{A}$, and taking the lower triangular component $\mathbf{L}$, the number of triangles is the reduction of the matrix $\mathbf{B} = \mathbf{L}\mathbf{L}^T .* \mathbf{L}$ to a scalar. Rows of $\mathbf{L}$ are sorted by increasing number of nonzeros, following the literature that demonstrates the benefits of sorting by degree prior to triangle counting [23]. In our implementation, we use a generalization of this algorithm where we assume we are solving the problem for three distinct matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{M}$ by computing $\mathbf{C} = \mathbf{A}\mathbf{B} .* \mathbf{M}$. We use the masked SpGEMM primitive whose implementation is detailed in Section 6.3.4. This is followed by a reduction of matrix $\mathbf{C}$ to a scalar, returning the number of triangles in graph $\mathbf{A}$. For each nonzero entry in the mask $\mathbf{M}(i, j) \neq 0$, our masked matrix-matrix multiplication performs an intersection of the nonzeros in the i-th row of $\mathbf{A}$ with the j-th column of $\mathbf{B}$. Using a straightforward merge-based set intersection would have yielded an implementation with $O(|E|^{3/2})$ work and $O(\lg^{3/2}|V|)$ depth [77]. Instead of performing the set intersection using merging or hash tables, we use repeated binary searches from the elements of the shorter list to the larger list. Multiple publications concluded that the method of repeated binary searches was either competitive with or faster than the merge-based method on GPUs [36, 44]. Theoretically, it increases work marginally by a factor $\log d$ on average where d is the average degree of a vertex. On the positive side, it has more parallelism.

8 EXPERIMENTAL RESULTS

We first show overall performance analysis of GraphBLAST on nine datasets including both real-world and synthetically generated graphs; the topology of these datasets spans from regular to

Dataset	Vertices	Edges	Max Degree	Diameter	Type
soc-orkut	3M	212.7M	27,466	9	rs
soc-Livejournal1	4.8M	85.7M	20,333	16	rs
hollywood-09	1.1M	112.8M	11,467	11	rs
indochina-04	7.4M	302M	256,425	26	rs
rmat_s22_e64	4.2M	483M	421,607	5	gs
rmat_s23_e32	8.4M	505.6M	440,396	6	gs
rmat_s24_e16	16.8M	519.7M	432,152	6	gs
rgg_n_24	16.8M	265.1M	40	2622	gm
roadnet_USA	23.9M	577.1M	9	6809	rm
coAuthorsCiteseer	227K	1.63M	1372	31*	rs
coPapersDBLP	540K	30.6M	3299	18*	rs
cit-Patents	3.77M	33M	793	24*	rs
com-Orkut	3.07M	234M	33313	8*	rs
road_central	14.1M	33.9M	8	4343*	rm
Journals	124	12K	123	2	rs
G43	1K	20K	36	4	gs
ship_003	122K	3.8M	143	58*	rs
belgium_osm	1.4M	3.1M	10	1923*	rm
roadNet-CA	2M	5.5M	12	617*	rm
delaunay_24	16.8M	101M	26	1720*	rm

Table 11. Dataset Description Table. Graph types are: r: real-world, g: generated, s: scale-free, and m: mesh-like. All datasets have been converted to undirected graphs. Self-loops and duplicated edges are removed. Datasets in the top segment are used for BFS, SSSP and PR. Datasets in the middle segment are used for TC. Datasets in the bottom segment are used for comparison with GBTL [88]. An asterisk indicates the diameter is estimated using samples from 10,000 vertices.

scale-free. Five additional datasets are used specifically for triangle counting, because they are the ones typically used for comparison of triangle counting [12, 81].

Measurement methodology. We report both runtime and traversed edges per second (TEPS) as our performance metrics. In general we report runtimes in milliseconds and TEPS as millions of traversals per second [MTEPS]. Runtime is measured by measuring the GPU kernel running time and TEPS is computed by the number of edges in the undirected graph divided by the runtime. We do not compute TEPS for CC, because it is not well-defined for this algorithm due to the hooking and pointer-jumping.

Hardware characteristics. We ran all experiments in this paper on a Linux workstation with 2×3.50 GHz Intel 4-core, hyperthreaded E5-2637 v2 Xeon CPUs, 528 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. GPU programs were compiled with NVIDIA's nvcc compiler (version 8.0.44) with the -O3 flag. CuSha was compiled using commit e753734 on their GitHub page. Galois was compiled using v2.2.1 (r0). Ligra was compiled using icpc 15.0.1 with CilkPlus at v1.5. Mapgraph was compiled at v0.3.3. SuiteSparse was compiled at v3.0.1 (beta1). Enterprise was compiled at commit 426846f on their GitHub page. Gunrock was compiled at v0.4 for the BFS, SSSP, PR and TC comparisons, and at v0.5 for the CC comparison. All results ignore transfer time (both disk-to-memory and CPU-to-GPU). All Gunrock and GraphBLAST tests were run 10 times with the average runtime and MTEPS used for results.

		Runtime (ms) [lower is better]					Edge throughput (MTEPS) [higher is better]				
Alg.	Dataset	SuiteSparse GraphBLAS	Hardwired GPU	Ligra	Gunrock	GraphBLAST	SuiteSparse GraphBLAS	Hardwired GPU	Ligra	Gunrock	GraphBLAST
BFS	soc-ork	2542	25.81	26.1	**5.573**	7.230	83.66	12360	8149	**38165**	29217
	soc-lj	2218	36.29	42.4	**14.05**	14.16	38.61	5661	2021	**6097**	6049
	h09	1013	11.37	12.8	**5.835**	7.138	111.1	14866	8798	**19299**	15775
	i04	2646	**67.7**	157	77.21	80.37	112.7	**8491**	1899	3861	3709
	rmat-22	5401	41.81	22.6	**3.943**	4.781	89.44	17930	21374	**122516**	101038
	rmat-23	8628	59.71	45.6	**7.997**	8.655	58.61	12971	11089	**63227**	58417
	rmat-24	21032	270.6	89.6	16.74	**16.59**	24.71	1920	5800	31042	**31327**
	rgg	230602	**138.6**	918	593.9	2991	1.201	**2868**	288.8	466.4	92.59
	road_usa	9413	**141**	978	676.2	7155	6.131	**1228**	59.01	85.34	8.065
SSSP	soc-ork	7223	807.2	**595**	981.6	676.7	29.45	263.5	**357.5**	216.7	314.3
	soc-lj	3599	369	368	393.2	**256.3**	23.81	232.2	232.8	217.9	**334.2**
	h09	2585	143.8	164	**83.2**	109.123	43.58	783.4	686.9	**1354**	1032
	i04	1087	—	397	**371.8**	414.5	274.22	—	750.8	**801.7**	719.2
	rmat-22	30688	—	774	583.9	**477.5**	15.74	—	624.1	827.3	**1011.7**
	rmat-23	25268	—	1110	739.1	**680.0**	20.01	—	455.5	684.1	**743.6**
	rmat-24	39105	—	1560	**884.5**	905.2	13.29	—	333.1	**587.5**	574.0
	rgg	1649653	—	**80800**	115554	14429l	0.161	—	**3.28**	2.294	1.84
	road_usa	801311	**4860**	29200	11037	144962	0.072	**11.87**	1.98	5.229	0.398
PR (pull)	soc-ork	942.3	**52.54**	476	173.1	64.22	225.7	**4048**	446.8	1229	3312
	soc-lj	741.8	33.61	200	54.1	**21.54**	115.5	2550	428.5	1584	**3978**
	h09	306.2	34.71	77.4	20.05	**8.12**	41.81	368.8	165.4	638.4	**1577**
	i04	1154	164.6	210	41.59	**19.16**	261.6	1835	1438	7261	**15763**
	rmat-22	5328	188.5	1250	304.5	**115.6**	90.65	2562	386.4	1586	**4178**
	rmat-23	7087	**147**	1770	397.2	161.3	71.34	**3439**	285.6	1273	3134
	rmat-24	9033	**128**	2180	493.2	211.5	57.54	**4060**	238.4	1054	2457
	rgg	2233	53.93	247	181.3	**34.58**	118.7	4916	1073	1462	**7665**
	road_usa	3030	—	209	**24.11**	26.91	190.4	—	2761	**23936**	21449
CC	soc-ork	34813	**46.97**	260	179.24	275.87	—	—	—	—	—
	soc-lj	32051	**43.51**	184	81.24	185.06	—	—	—	—	—
	h09	15551	**24.63**	90.8	92.1	64.44	—	—	—	—	—
	i04	57211	**130.3**	315	786.83	343.64	—	—	—	—	—
	rmat-22	81021	**149.4**	563	369.23	365.75	—	—	—	—	—
	rmat-23	89421	**212**	1140	498.01	705.45	—	—	—	—	—
	rmat-24	102867	**245.7**	1730	560.9	978.64	—	—	—	—	—
	rgg	190083	**103.9**	6000	353.41	5602	—	—	—	—	—
	road_usa	264328	**124.9**	50500	212.62	26880	—	—	—	—	—
TC	coauthor	6.337	**2.2**	—	4.51	5.96	128.5	**370**	—	181	137
	copaper	**50.93**	64.4	—	197	246	**630**	498	—	163	130
	soc-lj	1221	**295**	490	896	1125	56.5	**234**	141	77.0	61.3
	cit-pat	380.9	**34.5**	79.5	156	137	43.3	**478**	208	105	121
	com-ork	5100	**1626**	1920	6636	5367	23	**72.1**	61.0	17.7	21.8
	road_cent	231.3	**5.6**	—	61.4	78.7	73.1	**3018**	—	275	215

Table 12. GraphBLAST's performance comparison for runtime and edge throughput with other graph libraries (SuiteSparse, Ligra, Gunrock) and hardwired GPU implementations on a Tesla K40c GPU. All PageRank times are normalized to one iteration. Hardwired GPU implementations for each primitive are Enterprise (BFS) [55], delta-stepping SSSP [26], pull-based PR [51], hooking and pointer-jumping CC [79], and triangle counting [12]. A missing data entry means there is a runtime error.

Datasets. We summarize the datasets in Table 11. soc-orkut (soc-ork), com-orkut (com-ork), soc-Livejournal1 (soc-lj), and hollywood-09 (h09) are social graphs; indochina-04 (i04) is a crawled hyperlink graph from indochina web domains; coAuthorsCiteseer (coauthor), coPapersDBLP (copaper), and cit-Patents (cit-pat) are academic citation and patent citation networks; Journals (journal) is a graph indicating common readership across Slovenian magazines and journals; rmat_s22_e64 (rmat-22), rmat_s23_e32 (rmat-23), and rmat_s24_e16 (rmat-24) are three generated R-MAT graphs; and G43 (g43) is a random graph. All twelve datasets are scale-free graphs with diameters of less than 30 and unevenly distributed node degrees (80% of nodes have degree less than 64). ship-003 is a graph of a finite element model. The following datasets—rgg_n_24 (rgg), road_central (road_cent), roadnet_USA (road_usa), belgium_osm (belgium), roadNet-CA (road_ca), and delaunay_n24 (delaunay)—have large diameters with small and evenly distributed node degrees (most nodes have degree less than 12). soc-ork and com-Ork are from the Network Repository [68]; soc-lj, i04, h09, road_central, road_usa, coauthor, copaper, and cit-pat are from the University of Florida Sparse Matrix Collection [28]; rmat-22, rmat-23, rmat-24, and rgg are R-MAT and random geometric graphs we generated. The R-MAT graphs were generated with the following parameters: $a = 0.57$, $b = 0.19$, $c = 0.19$, $d = 0.05$. The edge weight values (used in SSSP) for each dataset are uniformly random integer values between 1 and 64.

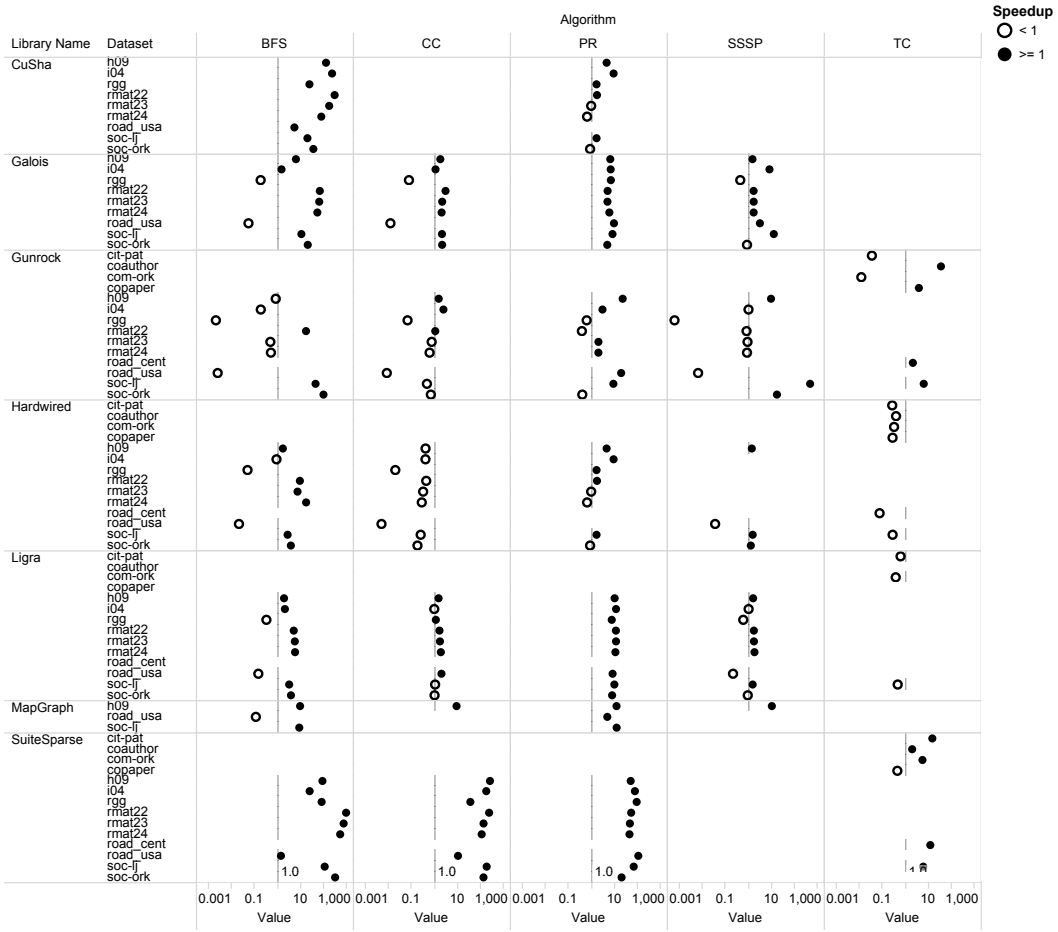


Fig. 11. Speedup of GraphBLAST over seven other graph processing libraries/hardwired algorithms on different graph inputs. Black dots indicate GraphBLAST is faster, white dots slower.

8.1 Performance summary

Table 12 and Figure 11 compare GraphBLAST's performance against several other graph libraries and hardwired GPU implementations. In general, GraphBLAST's performance on traversal-based algorithms (BFS and SSSP) is better on the seven scale-free graphs (soc-orkut, soc-lj, h09, i04, and rmats) than on the small-degree large-diameter graphs (rgg and road_usa). The main reason is our load-balancing strategy during traversal and particularly our emphasis on high performance for highly-skewed-distribution irregular graphs. Therefore, we incur a certain amount of overhead for our merge-based load-balancing and our requirement of a kernel launch in every iteration. For these types of graphs, asynchronous approaches, pioneered by Enterprise [55], that do not require exiting the kernel until the breakpoint has been met is a way to address the kernel launch problem. However, this does not work for non-BFS solutions, so asynchronous approaches in this area remain an open problem. In addition, graphs with uniformly low degree expose less parallelism and would tend to show smaller gains in comparison to CPU-based methods.

8.2 Comparison with CPU graph frameworks

We compare GraphBLAST's performance with three CPU graph libraries: the SuiteSparse GraphBLAS library, the first GraphBLAS implementation for multi-threaded CPU [27]; and Galois [62] and Ligra [76], both among the highest-performing multi-core shared-memory graph libraries. Against SuiteSparse, the speedup of GraphBLAST on average on all algorithms is geomean 27.9× (1268× peak) and geomean 43.51× (1268× peak) on scale-free graphs. Compared to Galois, GraphBLAST's performance is generally faster. We are 2.6× geomean (64.2× peak) faster across all algorithms. We get the greatest speedup on BFS, because we implement direction optimization. We get the next greatest speedup on PR, where the amount of computation tends to be greater than for BFS or SSSP.

Compared to Ligra, GraphBLAST's performance is generally comparable on most tested graph algorithms; note Ligra results are on a 2-CPU machine of the same timeframe as the K40c GPU we used to test. We are 3.38× (1.35× peak) faster for BFS vs. Ligra for scale-free graphs, because we incorporate some BFS-specific optimizations such as *masking*, *early-exit*, and *operand reuse*, as discussed in Section 7. However, we are 4.88× slower on the road network graphs. For SSSP, a similar picture emerges. Compared to Ligra for scale-free graphs, we get 1.35× (1.72× peak) speedup, but are 2.98× slower on the road networks. We believe this is because our Bellman-Ford with sparsification means we can do less work on scale-free graphs, but our framework is not optimized for road networks. For PR, we are 9.23× (10.96× peak) faster, because we use a highly-optimized merge-based load balancer that is suitable for this SpMV-based problem. For CC, we are 1.30× (1.88× peak) faster. With regards to TC, we are 2.80× slower, because we have a simple algorithm for the masked matrix-matrix multiply.

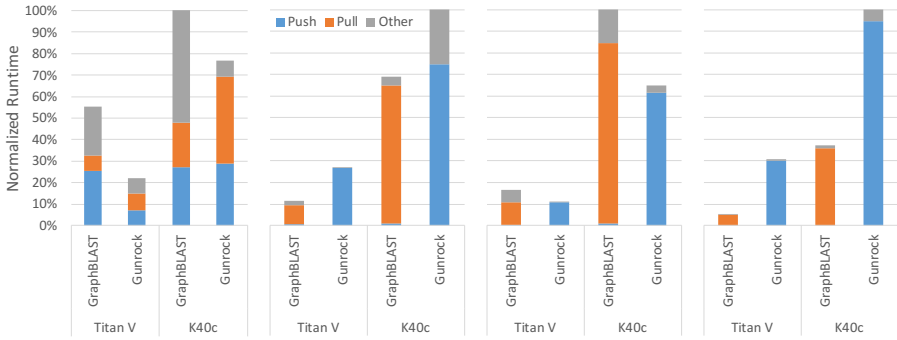
8.3 Comparison with GPU graph frameworks and GPU hardwired

Compared to hardwired GPU implementations, depending on the dataset, GraphBLAST's performance is comparable or better on BFS, SSSP, and PR. For CC, GraphBLAST is 3.1× slower (geometric mean) on scale-free graphs and 107.7× slower on road network graphs. We think the reason is that we are strictly following Shiloach-Vishkin in doing only one level of pointer jumping per iteration, whereas the hardwired implementation is doing pointer jumping until each tree has been reduced to a star. This is an advantage on the GPU, because it allows their implementation to significantly reduce kernel launch overheads (see Section 2.1.4), which become significant especially for graphs with high diameter such as road networks. If kernel fusion is added to GraphBLAST, we would be able to take advantage of this optimization.

For TC, GraphBLAST is 3.3× slower (geometric mean) than the hardwired GPU implementation due to fusing of the matrix-multiply and the reduce, which lets the hardwired implementation avoid the step of writing out the output to the matrix-multiply. The alternative is having a specialized kernel that does a fused matrix-multiply and reduce. This tradeoff is not typical of our other algorithms. While still achieving high performance, GraphBLAST's application code is smaller in size and clearer in logic compared to other GPU graph libraries.

Compared to CuSha and MapGraph, GraphBLAST's performance is quite a bit faster. We get geomean speedups of 8.40× and 3.97× respectively (420× and 64.2× peak). The speedup comes from direction optimization. CuSha only does the equivalent of pull-traversal, so their performance is most comparable to ours in PR. MapGraph is push-only.

Compared to Gunrock, the fastest GPU graph framework, GraphBLAST's performance is comparable on BFS, CC and TC with Gunrock being 11.8%, 14.8% and 11.1% faster in the geomean respectively. On SSSP, GraphBLAST is faster by 1.1× (1.53× peak). This can be attributed to GraphBLAST using *generalized direction optimization* and Gunrock only doing push-based advance. On PR, GraphBLAST is significantly faster and gets speedups of 2.39× (5.24× peak). For PR, the speed-up



(a) BFS (b) SSSP (c) CC (d) PR

Fig. 12. Runtime breakdown of GraphBLAST and Gunrock migrating from K40c to Titan V GPU for BFS, SSSP, CC and PR on 'soc-ork', normalized to the slowest combination per graph algorithm.

again can be attributed to GraphBLAST automatically using *generalized direction optimization* to select the right direction, which is SpMV in this case. Gunrock does push-based advance.

8.4 Comparison with Gunrock on latest GPU architecture

In Table 13, we compare against Gunrock on BFS, SSSP, CC and PR using the latest generation GPU, Titan V. As the result shows, we have a 3.13× slowdown compared to Gunrock on BFS, which indicates that we do worse on Titan V. On SSSP, we are 1.08× (2.39× peak) faster when not including the road network datasets, and 0.48× slower when including them. On CC, we have a 4.00× slowdown compared to Gunrock. On PR, we are 2.90× (7.67× peak) faster in the geomean.

Taking a closer look at this in Figure 12, we can see that both push and pull components of Gunrock's BFS benefit due to moving from K40c to Titan V, but "other" does not. However for GraphBLAST, only the "other" and pull benefit from moving from K40c to Titan V. We hypothesize the reason for this is the GraphBLAST push is implemented using a radixsort to perform a multiway merge, and radixsort does not see a noticeable improvement in performance from K40c to Titan V on the problem sizes typical of BFS. On the other hand, Gunrock uses uses a series of inexpensive heuristics [82] to reduce but not eliminate redundant entries in the output frontier. These heuristics include a global bitmask, a block-level history hashtable, and a warp-level hashtable. The size of each hashtable is adjustable to achieve the optimal tradeoff between performance and redundancy reduction rate. However, this approach may not be suitable for GraphBLAS, because such an optimization may be too BFS-focused and would generalize poorly.

For SSSP, GraphBLAST has the advantage of the direction optimization automatically choosing the optimal direction, which for SSSP happens to be pull. Gunrock's SSSP is written to use push, so it is unable to take advantage of this feature. For PR, the situation is similar to SSSP in that Gunrock uses push when pull is better. In CC, Gunrock's implementation has the advantage of following the approach of the hardwired implementation by Soman, Kishore and Narayanan [79]. This algorithm performs pointer-jumping until the tree has become a star, whereas GraphBLAST follows Shiloach-Vishkin strictly and only does one level of pointer-jumping. We believe Gunrock's primary performance advantage, then, is faster convergence. More research is required to understand whether this optimization can be used to improve the linear algebraic implementation of FastSV in the presence and absence of kernel fusion (see Section 2.1.4).

Alg.	Type	Dataset	Runtime (ms) [lower is better] Gunrock	Runtime (ms) [lower is better] GraphBLAST	Edge throughput (GTEPS) [higher is better] Gunrock	Edge throughput (GTEPS) [higher is better] GraphBLAST	Speedup	Geomean Speedup
BFS	Scale-free	soc-ork	1.61	4.02	132.3	52.93	0.40×	0.44×
		soc-lj	2.95	8.27	29.05	10.36	0.36×	
		h09	1.60	5.16	70.51	21.83	0.31×	
		i04	14.72	32.09	20.25	9.29	0.46×	
		rmat-22	1.13	2.36	425.7	204.6	0.48×	
		rmat-23	2.04	3.64	247.3	138.8	0.56×	
		rmat-24	3.91	6.79	132.8	76.47	0.58×	
	Road network	rgg	321.0	3333	0.863	0.083	0.096×	0.10×
		road_usa	782.5	7467	0.074	0.0077	0.10×	
SSSP	Scale-free	soc-ork	263.0	110.2	0.809	1.93	**2.39**×	1.08×
		soc-lj	122.5	72.41	0.699	1.18	**1.69**×	
		h09	15.46	43.87	7.29	2.57	0.35×	
		i04	79.01	150.8	3.77	1.98	0.52×	
		rmat-22	103.6	76.23	4.66	6.34	**1.35**×	
		rmat-23	175.5	122.8	2.88	4.12	**1.43**×	
		rmat-24	254.0	209.5	2.05	2.48	**1.21**×	
	Road network	rgg	349.6	91704	0.792	0.0030	0.0038×	0.025×
		road_usa	2928	17994	0.020	0.0032	0.16×	
PR (pull)	Scale-free	soc-ork	51.76	9.29	4.11	22.88	**5.57**×	2.86×
		soc-lj	14.66	4.41	5.84	19.45	**3.33**×	
		h09	3.99	2.11	28.23	53.40	**1.89**×	
		i04	6.54	5.38	45.58	55.37	**1.21**×	
		rmat-22	49.26	15.86	9.81	30.46	**3.11**×	
		rmat-23	85.62	24.23	5.91	20.87	**3.53**×	
		rmat-24	130.1	39.14	3.99	13.28	**3.32**×	
	Road network	rgg	62.17	8.10	4.46	34.18	**7.67**×	3.06×
		road_usa	9.17	7.51	6.29	7.68	**1.22**×	
CC	Scale-free	soc-ork	30.14	45.54	N/A	N/A	0.66×	0.68×
		soc-lj	15.91	47.7	N/A	N/A	0.33×	
		h09	18.05	18.3	N/A	N/A	0.99×	
		i04	65.55	105.87	N/A	N/A	0.62×	
		rmat-22	65.33	56.68	N/A	N/A	**1.15**×	
		rmat-23	87.9	117.79	N/A	N/A	0.76×	
		rmat-24	126.84	226.36	N/A	N/A	0.56×	
	Road network	rgg	64.25	4826.8	N/A	N/A	0.013 ×	0.0077×
		road_usa	39.78	9016.04	N/A	N/A	0.0044×	

Table 13. GraphBLAST's performance comparison for runtime and edge throughput with Gunrock [82] for four graph algorithms on a Titan V GPU. Datasets where this work is faster are shown in bold.

8.5 Comparison with GraphBLAS-like framework on GPU

In Table 14, we compare against GBTL [88], the first GraphBLAS-like implementation for the GPU on BFS. Our implementation is 31.8× (58.5× peak) faster in the geomean. We attribute this

Dataset	Runtime (ms) [lower is better]		Edge throughput (MTEPS) [higher is better]		Speedup
	GBTL	GraphBLAST	GBTL	GraphBLAST	
Journals	5.76	0.147	2.074	80.98	39.05×
G43	14.61	0.503	1.368	39.72	29.04×
ship_003	559.0	9.562	14.25	832.9	58.46×
belgium_osm	10502	476.3	0.295	6.508	22.05×
roadNet-CA	4726	259.2	1.168	21.30	18.23×
delaunay_24	65508	1677	1.537	60.02	39.06×

Table 14. GraphBLAST's performance comparison for runtime and edge throughput with GBTL [88] for BFS on a Tesla K40c GPU.

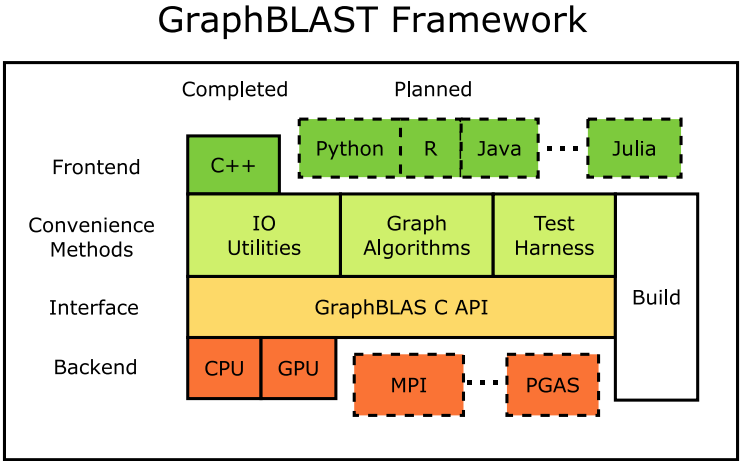


Fig. 13. Design of GraphBLAST: Completed and planned components, and how open standard GraphBLAS API fits into the framework.

speed-up to several factors: (1) they use the Thrust library [10] to manage the CPU-to-GPU memory traffic that works for all GPU applications, while we use a domain-specific memory allocator that only copies from CPU to GPU when necessary; (2) they specialize the CUSP library's `mxm` operation [24, 25] for a matrix with a single column to mimic the `mxv` required by the BFS, while we have a specialized `mxv` operation that is more efficient; and (3) we utilize the design principles of exploiting input and output sparsity, as well as proper load balancing, none of which are in GBTL.

In addition to getting comparable or faster performance, GraphBLAST has the advantage of being concise, as shown in Table 1. Developing new graph algorithms in GraphBLAST requires modifying a single file and writing straightforward C++ code. Currently, we are working on a Python frontend interface too, to allow users to build new graph algorithms without having to recompile. Additional language bindings are being planned as well (see Figure 13). Similar to working with machine learning frameworks, writing GraphBLAST code does not require any parallel programming knowledge of OpenMP, OpenCL or CUDA, or even performance optimization experience.

9 CONCLUSION

In this paper, we set out to answer the question: What is needed for a high-performance graph framework that is based on linear algebra? The answer we conclude is that it must: (1) exploit input sparsity through direction optimization, (2) exploit output sparsity through masking, and (3) address the GPU considerations of avoiding CPU-to-GPU copies, supporting generalized semiring operators, and load-balancing. In order to give empirical evidence for this hypothesis, using the above design principles we built a framework called GraphBLAST based on the GraphBLAS open standard. Testing GraphBLAST on five graph algorithms, we were able to obtain 36× geomean 892× peak) over SuiteSparse GraphBLAS (sequential CPU) and 2.14× geomean (10.97× peak) and 1.01× (5.24× peak) speed-up over Ligra and Gunrock respectively, which are state-of-the-art graph frameworks on the CPU and GPU.

What follows are the main limitations of GraphBLAST: *multi-GPU and multi-node scaling*, *kernel fusion*, *asynchronous execution*, and *matrix-matrix generalization of direction optimization*. As evidenced by other algorithms listed in its code repository, GraphBLAST supports a much larger set of algorithms than the ones described in this paper. In fact, given that our work implements the GraphBLAS API with only minor modifications, it potentially supports the growing list of algorithms that are implemented using GraphBLAS, a majority of which are included in the LAGraph repository[3]. The only graph algorithms that do not map to the GraphBLAS API efficiently are those that are inherently sequential such as depth-first search and algorithms that use priority queues such as Dijkstra's algorithm.

Multi-GPU and multi-node scalability. By construction, the GraphBLAS open standard establishes its first two goals—*portable performance* and *conciseness*. Portable performance is from making implementers adhere to the same standard interface; conciseness, by basing the interface design around the language of mathematics, which is one of the most concise forms of expression. In this paper, we set out to meet the third goal of *high performance*, which is a prerequisite towards *scalability*. Our work does not address scalability. While we have demonstrated that GraphBLAS is effective at the scale of a single GPU, we have not addressed the issues associated with scaling across multiple GPUs much less multiple nodes.

The largest challenge for the scale-up direction (more capable nodes) is the limited size of GPU main memory. The NVIDIA K40c (and V100) we used in our experiments, for instance, only support 12 GB (32 GB) of main memory, and all of the datasets we have used in our experiments fit into a 12 GB memory allocation. CPUs support a much larger main memory, allowing CPU-based systems like Ligra to scale their performance to much larger datasets on a single processor. For graph applications that run on GPUs but use CPU memory to store a (larger) graph, the low bandwidth between CPU and GPU and the generally irregular access patterns into the graph data structure would likely make using the CPU for backing storage noncompetitive vs. datasets that fit into GPU memory.

The largest challenge for the scale-out direction (more nodes) is effectively and quickly partitioning scale-free graphs. Road-network-like graphs partition well, resulting in a tractable amount of communication between partitions, and thus would generally be amenable to scaling across multiple GPUs. But, because of their high connectivity, scale-free graphs are much more difficult to partition. The resulting high communication volume makes scalability much more difficult.

GPU-based implementations have found difficulty in scaling to as many nodes as CPU-based implementations. This is partly due to GPUs speeding up each node's local computation phase, thus increasing the algorithm's sensitivity to any latency from inter-node communication; and

[3] www.github.com/GraphBLAS/LAGraph

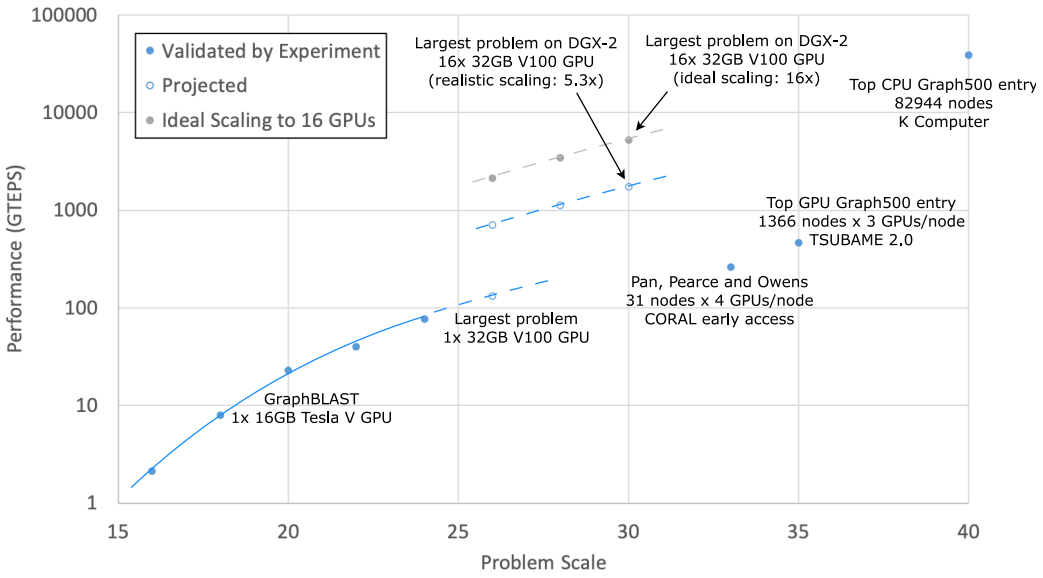


Fig. 14. Data points from GraphBLAST and points representative of the state-of-the-art in distributed BFS. The Dashed line indicates projected performance assuming ideal scaling and realistic scaling accounting for bisection bandwidth for 16 GPUs. In random graph generation for each problem scale *SCALE*, the graph will have 2^{SCALE} vertices and 16×2^{SCALE} edges according to Graph500 rules. We acknowledge that the R-MAT generator used by Graph500 has known densification issues [73] that make weak scaling studies problematic. However, Graph500 remains the community standard for benchmarking weakly scaled graph studies.

partly because each GPU has very limited main memory compared to CPUs. New GPU-based fat nodes such as the DGX-2 may offer an interesting solution to both problems. By offering 16× GPUs with 32 GB memory each and by being connected using NVSwitch technology that offers a bisection bandwidth of 2.4 TB/s, the DGX-2 may be a contender for multi-GPU top BFS performance. For example, in Figure 14, the dashed line and hollow point indicate the potential performance of a DGX-2 system, assuming linear scalability from the 1× GPU GraphBLAST BFS and realistic scalability given the bisection bandwidth computation as follows:

The V100 has a memory bandwidth of 900 GB/s. The BFS on scale-free graphs that are challenging to partition have $O(|E|/P)$ bandwidth cost, where P is the number of devices (such as GPUs), regardless of the algorithm used [16]. Given half of the processors will be on each side of the bisection, $O(|E|/2)$ data will need to be exchanged. If a single V100 was used, per-edge bandwidth would be $900/|E|$, because we need to touch each edge at some point. With 16 V100s, it is $4800/|E|$, so a more realistic speedup is 5.3× faster on the DGX-2 compared to a single V100.

Kernel fusion. In this paper, we hinted at several open problems as potential directions of research. One open problem is the problem of kernel fusion (Section 2.1.4). In the present situation, a GraphBLAS-based triangle counting algorithm in *blocking mode* (i.e. where operations are required to complete before the next operation begins) can never be as efficient as a hardwired GPU implementation, because it requires a matrix-matrix multiply followed by a reduce. This bulk-synchronous approach forces the computer to write the output of the matrix-matrix multiply to main memory before reading from main memory again in the reduce. A worthwhile area of programming language research would be to use a computation graph to store the operations that

must happen, do a pass over the computation graph to identify profitable kernels to fuse, generate the CUDA kernel code at runtime, just-in-time (JIT) compile the code to machine code, and execute the fused kernel. This may be possible in GraphBLAS's non-blocking mode where operations are not required to return immediately after each operation, but only when the user requests an output or an explicit `wait`.

Such an approach is what is done in machine learning, but with graph algorithms the researcher is faced with additional challenges. One such challenge is that the runtime of graph kernels is dependent on the input data, so in a multiple iteration algorithm such as BFS, SSSP or PR, it may be profitable to fuse two kernels in one iteration and two different kernels in a different iteration. Another challenge is the problem of load balancing. Typically code that is automatically generated is not as efficient as hand-tuned kernels, and may not load-balance well enough to be efficient.

Asynchronous execution model. For road network graphs, asynchronous approaches pioneered by Enterprise [55] that do not require exiting the kernel until the breakpoint has been met is a way to address the kernel launch problem. This opens the door to two avenues of research: (1) How can one detect whether one is dealing with a road network that will require thousands of iterations to converge rather than tens of iterations? (2) How can such an asynchronous execution model be reconciled with GraphBLAS, which is based on the bulk-synchronous parallel model? The first problem requires a system that detects whether the graph is chordal, planar, bipartite, etc. before running the graph traversal algorithm, while the latter problem may also have implications when scaling to distributed implementations.

Matrix-matrix generalization of direction optimization. Currently, direction optimization is only applied for matrix-vector multiplication. However, in the future, the optimization can be extended to matrix-matrix multiplication. The analogue is thinking of the matrix on the right as not a single vector, but as composed of many column vectors, each representing a graph traversal from a different source node. Applications include batched betweenness centrality and all-pairs shortest-path. Instead of switching between SpMV and SpMSpV, we could be switching between SpMM (sparse matrix-dense matrix) and SpGEMM (sparse matrix-sparse matrix).

Finally, several statistical algorithms can estimate the size and structure of SpGEMM output [1, 22], which can be used to choose the right algorithm when implementing direction-optimizing matrix-matrix multiplication.

ACKNOWLEDGMENTS

We thank Yuechao Pan for valuable insight into BFS optimizations. We would like to acknowledge Scott McMillan for important feedback on early drafts of the paper. Thanks to Olivier Beaumont for the historical background on push-pull terminology. We thank Muhammad Osama and Charles Rozhon for collecting valuable experimental results for Gunrock. We would like to acknowledge Collin McCarthy for maintaining the servers the experiments were run on in top shape. We appreciate funding from the National Science Foundation (Award # CCF-1629657 and OAC-1740333), the DARPA XDATA, HIVE and SDH programs, and for support from an NVIDIA AI Laboratory (UC Davis Center for GPU Graph Analytics). This material is based on research sponsored by Air Force Research Lab (AFRL), the Army Research Lab (ARL), and the Defense Advanced Research Projects Agency (DARPA) under agreements FA8650-18-2-7836, W911QX-12-C-0059 and HR0011-18-3-0007.

This research was supported in part by the Applied Mathematics program of the DOE Office of Advanced Scientific Computing Research under Contract No. DE-AC02-05CH11231, and in part the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, ARL, DARPA, or the U.S. Government.

REFERENCES

[1] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. *Lecture Notes in Computer Science*, pages 406–419, 2010.

[2] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, May 2015.

[3] Omar Batarfi, Radwa El Shawi, Ayman G. Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi, and Sherif Sakr. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, July 2015.

[4] Sean Baxter. Modern GPU library. https://moderngpu.github.io/, 2016.

[5] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[6] Scott Beamer. *Understanding and Improving Graph Algorithm Performance*. PhD thesis, University of California, Berkeley, Fall 2016.

[7] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, November 2012.

[8] Scott Beamer, Krste Asanović, and David Patterson. Reducing PageRank communication via propagation blocking. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831, May 2017.

[9] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, SC '09, pages 18:1–18:11, November 2009.

[10] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems Jade Edition*, pages 359–371. Elsevier, 2012.

[11] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104. ACM, June 2017.

[12] Mauro Bisson and Massimiliano Fatica. High performance exact triangle counting on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3501–3510, December 2017.

[13] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990. http://www.cs.cmu.edu/~scandal/papers/CMU-CS-90-190.html.

[14] Benjamin Brock, Aydin Buluç, Timothy G. Mattson, Scott McMillan, and José E. Moreira. A roadmap for the GraphBLAS C++ API. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 219–222. IEEE, May 2020.

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

[16] Aydın Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM Press, November 2011.

[17] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. *The GraphBLAS C API Specification*, November 2017. Rev. 1.1. http://graphblas.org/index.php/C_language_API.

[18] Aydın Buluç, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017.

[19] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, September 2018.

[20] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 85–98, 2012.

[21] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, August 2015.

[22] Edith Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1998.
[23] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science & Engineering*, 11(4):29–41, July 2009.
[24] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0. http://cusplibrary.github.io/.
[25] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix-matrix multiplication for the GPU. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25:1–25:20, August 2015.
[26] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2014, pages 349–359, May 2014.
[27] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software*, 45(4):1–25, December 2019.
[28] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1:1–1:25, November 2011.
[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.
[30] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer Berlin Heidelberg, 2009.
[31] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934, 2019.
[32] Niels Doekemeijer and Ana Lucia Varbanescu. A survey of parallel graph processing frameworks. Technical Report PDS-2014-003, Delft University of Technology, 2014.
[33] Joe Eaton. nvGRAPH. https://docs.nvidia.com/cuda/nvgraph/index.html, 2016. Accessed: 2018-01-18.
[34] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, 2012.
[35] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):30:1–30:49, March 2017.
[36] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A. Bader. Fast and adaptive list intersections on the GPU. *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, September 2018.
[37] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of the Workshop on GRAph Data Management Experiences and Systems*, GRADES '14, pages 2:1–2:6, June 2014.
[38] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. Extreme scale de novo metagenome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '18, pages 10:1–10:13. ACM/IEEE, November 2018.
[39] D. Gleich, L. Zhukov, and P. Berkhin. Fast parallel PageRank: a linear system approach. Technical Report YRL-2004-038, Yahoo! Research, 2004.
[40] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, OSDI '12, pages 17–30. USENIX Association, October 2012.
[41] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, September 1978.
[42] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of Pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, August 2014.
[43] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.
[44] Yang Hu, Hang Liu, and H. Howie Huang. TriCore: Parallel triangle counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.
[45] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine*

Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 2323–2332, Stockholmsmässan, Stockholm Sweden, July 2018.

[46] Ben Johnson, Weitang Liu, Agnieszka Łupińska, Muhammad Osama, John D. Owens, Yuechao Pan, Leyuan Wang, Xiaoyun Wang, and Carl Yang. HIVE year 1 report: Executive summary. https://gunrock.github.io/docs/hive_year1_summary.html, November 2018.

[47] David S. Johnson and Catherine C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12. American Mathematical Society, 1993.

[48] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. Adaptive methods for the computation of PageRank. *Linear Algebra and its Applications*, 386:51–65, July 2004. Special Issue on the Conference on the Numerical Solution of Markov Chains 2003.

[49] R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 565–574, November 2000.

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

[51] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, June 2014.

[52] Michael Kircher and Prashant Jain. Pooling. In Alan O'Callaghan, Jutta Eckstein, and Christa Schwanninger, editors, *Proceedings of the 7th European Conference on Pattern Languages of Programms*, EuroPLoP 2002, pages 497–510. UVK—Universitätsverlag Konstanz, July 2002.

[53] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, OSDI '12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[54] Dénes Kőnig. Gráfok és mátrixok (graphs and matrices). *Matematikai és Fizikai Lapok*, 38:116–119, 1931. English translation by Gábor Szárnyas, Sept. 2020, https://arxiv.org/abs/2009.03780.

[55] Hang Liu and H. Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 68:1–68:12, November 2015.

[56] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, June 2010.

[57] Timothy G. Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E. Moreira. GraphBLAS C API: Ideas for future versions of the specification. In *IEEE High Performance Extreme Computing Conference (HPEC)*, September 2017.

[58] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2):1–39, November 2015.

[59] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 678–689, November 2016.

[60] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, February 2012.

[61] Jose Moreira and Bill Horn. IBM GraphBLAS. http://github.com/IBM/ibmgraphblas, 2018.

[62] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 456–471. ACM Press, November 2013.

[63] Muhammad Osama, Minh Truong, Carl Yang, Aydın Buluç, and John D. Owens. Graph coloring on the GPU. In *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning*, GrAPL 2019, pages 231–240, May 2019.

[64] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydın Buluç. Terrace: A hierarchical graph container for skewed dynamic graphs. In *SIGMOD*, 2021.

[65] Josh Patterson. RAPIDS: Open GPU data science. https://rapids.ai/, 2018.

[66] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 549–559. IEEE, November 2014.

[67] Roger Pearce, Trevor Steil, Benjamin W. Priest, and Geoffrey Sanders. One quadrillion triangles queried on one million processors. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, September 2019.

[68] Ryan Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4292–4293, January 2015.

[69] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 472–488. ACM Press, November 2013.

[70] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. GraphChallenge.org: Raising the bar on graph analytic performance. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, September 2018.

[71] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. Efficient parallel scan algorithms for many-core GPUs. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*, Chapman & Hall/CRC Computational Science, chapter 19, pages 413–442. Taylor & Francis, January 2011.

[72] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, August 2007.

[73] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. An in-depth study of stochastic Kronecker graphs. In *2011 IEEE 11th International Conference on Data Mining*, pages 587–596. IEEE, December 2011.

[74] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on GPUs: A survey. *ACM Computing Surveys*, 50(6):1–35, January 2018.

[75] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, March 1982.

[76] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, February 2013.

[77] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160. IEEE, April 2015.

[78] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, May 2014.

[79] Jyothish Soman, Kothapalli Kishore, and P J Narayanan. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, April 2010.

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

[81] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A comparative study on exact triangle counting algorithms on the GPU. In *Proceedings of the 1st High Performance Graph Processing Workshop*, HPGP '16, pages 1–8, May 2016.

[82] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):3:1–3:49, August 2017.

[83] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, September 2017.

[84] Carl Yang, Aydın Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the GPU. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Proceedings of the IEEE International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 672–687, August 2018.

[85] Carl Yang, Aydın Buluç, and John D. Owens. Implementing push-pull efficiently in GraphBLAS. In *Proceedings of the International Conference on Parallel Processing*, ICPP 2018, pages 89:1–89:11, August 2018.

[86] Carl Yang, Yangzihao Wang, and John D. Owens. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *Graph Algorithms Building Blocks*, GABB 2015, pages 841–847, May 2015.

[87] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. Understanding the overheads of launching CUDA kernels. In *Proceedings of the International Conference on Parallel Processing, Poster Session*, ICPP 2019, August 2019.

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

[89] Yongzhe Zhang, Ariful Azad, and Aydın Buluç. Parallel algorithms for finding connected components using linear algebra. *Journal of Parallel and Distributed Computing*, 144:14–27, October 2020.

[90] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. FastSV: A distributed-memory connected component algorithm with fast convergence. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, PP20, pages 46–57. SIAM, February 2020.

A LINES OF CODE

For the purposes of counting lines of code, we only compare the code specific to each algorithm, and we assume the graph data structure is already stored in the preferred format by each framework. We compare the lines of code by manually deleting the include and namespace statements, running the open-source code formatting tool `clang-format` using the default options and then using the open-source tool `cloc` to count the numbers of lines of code. This process is used to have as fair a comparison as possible, because some frameworks require many include and namespace statements, while others have only a few. Similarly, some codebases exceed the 80 character line limit, whereas others respect it. By using `clang-format` and `cloc`, we can sidestep many of these issues. In this section, we compare our framework against the non-GraphBLAS-based framework with the fewest lines of code, which happens to be Ligra [76] for all five algorithms.

A.1 Breadth-first-search

```
void bfs(Vector<float> *v, const Matrix<float> *A, Index s, Descriptor *desc) {
  Index A_nrows;
  CHECK(A->nrows(&A_nrows));
  CHECK(v->fill(0.f));
  Vector<float> q1(A_nrows);
  Vector<float> q2(A_nrows);
  std::vector<Index> indices(1, s);
  std::vector<float> values(1, 1.f);
  CHECK(q1.build(&indices, &values, 1, GrB_NULL));
  float iter = 1;
  float succ = 0.f;
  do {
    assign<float, float>(v, &q1, GrB_NULL, iter, GrB_ALL, A_nrows, desc);
    CHECK(desc->toggle(GrB_MASK));
    vxm<float, float, float, float>(
        &q2, v, GrB_NULL, LogicalOrAndSemiring<float>(), &q1, A, desc);
    CHECK(desc->toggle(GrB_MASK));
    CHECK(q2.swap(&q1));
    reduce<float, float>(&succ, GrB_NULL, PlusMonoid<float>(), &q1, desc);
    iter++;
  } while (succ > 0);
}
```

```
template <class ET> inline void writeOr(ET *a, ET b) {
  volatile ET newV, oldV;
  do {
    oldV = *a;
    newV = oldV | b;
  } while ((oldV != newV) && !CAS(a, oldV, newV));
}
struct BFS_F {
  uintE *Parents;
  long *Visited;
  BFS_F(uintE *_Parents, long *_Visited)
      : Parents(_Parents), Visited(_Visited) {}
  inline bool update(uintE s, uintE d) {
    writeOr(&Visited[d / 64], Visited[d / 64] | ((long)1 << (d \% 64)));
    Parents[d] = s;
    return 1;
  }
  inline bool updateAtomic(uintE s, uintE d) {
    writeOr(&Visited[d / 64], Visited[d / 64] | ((long)1 << (d \% 64)));
    return Parents[d] == UINT_E_MAX && CAS(&Parents[d], UINT_E_MAX, s);
  }
  inline bool cond(uintE d) {
    return !(Visited[d / 64] & ((long)1 << (d \% 64)));
  }
};
template <class vertex> void Compute(graph<vertex> &GA, commandLine P) {
  long start = P.getOptionLongValue("-r", 0);
  long n = GA.n;
  uintE *Parents = newA(uintE, n);
  parallel_for(long i = 0; i < n; i++) Parents[i] = UINT_E_MAX;
  Parents[start] = start;
  long numWords = (n + 63) / 64;
  long *Visited = newA(long, numWords);
  { parallel_for(long i = 0; i < numWords; i++) Visited[i] = 0; }
  Visited[start / 64] = (long)1 << (start \% 64);
  vertexSubset Frontier(n, start);
  while (!Frontier.isEmpty()) {
    vertexSubset output = edgeMap(GA, Frontier, BFS_F(Parents, Visited));
    Frontier.del();
    Frontier = output;
  }
  Frontier.del();
  free(Parents);
  free(Visited);
}
```

Algorithm 2. BFS code for GraphBLAST (left), Ligra [76] (right)

A.2 Single-source shortest-path

```
void sssp(Vector<float> *v, const Matrix<float> *A, Index s, Descriptor *desc) {
  Index A_nrows;
  A->nrows(&A_nrows);
  std::vector<graphblas::Index> indices(1, s);
  std::vector<float> values(1, 0.f);
  v->build(&indices, &values, 1, GrB_NULL);
  Vector<float> w(A_nrows);
  Vector<float> zero(A_nrows);
  zero.fill(std::numeric_limits<float>::max());
  Index iter = 1;
  float succ_last = 0.f;
  float succ = 1.f;
  do {
    succ_last = succ;
    vxm<float, float, float, float>(
        &w, GrB_NULL, GrB_NULL, MinimumPlusSemiring<float>(), v, A, desc);
    eWiseAdd<float, float, float, float>(
        v, GrB_NULL, GrB_NULL, MinimumPlusSemiring<float>(), v, &w, desc);
    eWiseMult<float, float, float, float>(
        &w, GrB_NULL, GrB_NULL, PlusLessSemiring<float>(), v, &zero, desc);
    reduce<float, float>(&succ, GrB_NULL, PlusMonoid<float>(), &w, desc);
    iter++;
  } while (succ_last != succ);
}
```

```
#define WEIGHTED 1
struct BF_F {
  intE *ShortestPathLen;
  int *Visited;
  BF_F(intE *_ShortestPathLen, int *_Visited)
      : ShortestPathLen(_ShortestPathLen), Visited(_Visited) {}
  inline bool update(uintE s, uintE d, intE edgeLen) {
    intE newDist = ShortestPathLen[s] + edgeLen;
    if (ShortestPathLen[d] > newDist) {
      ShortestPathLen[d] = newDist;
      if (Visited[d] == 0) {
        Visited[d] = 1;
        return 1;
      }
    }
    return 0;
  }
  inline bool updateAtomic(uintE s, uintE d, intE edgeLen) {
    intE newDist = ShortestPathLen[s] + edgeLen;
    return (writeMin(&ShortestPathLen[d], newDist) && CAS(&Visited[d], 0, 1));
  }
  inline bool cond(uintE d) { return cond_true(d); }
};
struct BF_Vertex_F {
  int *Visited;
  BF_Vertex_F(int *_Visited) : Visited(_Visited) {}
  inline bool operator()(uintE i) {
    Visited[i] = 0;
    return 1;
  }
};
template <class vertex> void Compute(graph<vertex> &GA, commandLine P) {
  long start = P.getOptionLongValue("-r", 0);
  long n = GA.n;
  intE *ShortestPathLen = newA(intE, n);
  { parallel_for(long i = 0; i < n; i++) ShortestPathLen[i] = INT_MAX / 2; }
  ShortestPathLen[start] = 0;
  int *Visited = newA(int, n);
  { parallel_for(long i = 0; i < n; i++) Visited[i] = 0; }
  vertexSubset Frontier(n, start);
  long round = 0;
  while (!Frontier.isEmpty()) {
    if (round == n) {
      {
        parallel_for(long i = 0; i < n; i++) ShortestPathLen[i] =
            -(INT_E_MAX / 2);
      }
      break;
    }
    vertexSubset output = edgeMap(GA, Frontier, BF_F(ShortestPathLen, Visited),
                                  GA.m / 20, dense_forward);
    vertexMap(output, BF_Vertex_F(Visited));
    Frontier.del();
    Frontier = output;
    round++;
  }
  Frontier.del();
  free(Visited);
  free(ShortestPathLen);
}
```

Algorithm 3. SSSP code for GraphBLAST (left), Ligra [76] (right)

A.3 PageRank

```
void pr(Vector<float> *p, const Matrix<float> *A, float alpha, float eps,
        Descriptor *desc) {
  Index A_nrows;
  CHECK(A->nrows(&A_nrows));
  CHECK(p->clear());
  CHECK(p->fill(1.f / A_nrows));
  Vector<float> p_prev(A_nrows);
  Vector<float> p_swap(A_nrows);
  Vector<float> r(A_nrows);
  r.fill(1.f);
  Vector<float> r_temp(A_nrows);
  float error_last = 0.f;
  float error = 1.f;
  for (int iter = 1; error > eps && iter <= desc->descriptor_.max_niter_;
       ++iter) {
    error_last = error;
    p_prev = *p;
    vxm<float, float, float, float>(&p_swap, GrB_NULL, GrB_NULL,
                                    PlusMultipliesSemiring<float>(), &p_prev, A,
                                    desc);
    eWiseAdd<float, float, float, float>(
        p, GrB_NULL, GrB_NULL, PlusMultipliesSemiring<float>(), &p_swap,
        (1.f - alpha) / A_nrows, desc);
    eWiseMult<float, float, float, float>(
        &r, GrB_NULL, GrB_NULL, PlusMinusSemiring<float>(), p, &p_prev, desc);
    eWiseAdd<float, float, float, float>(&r_temp, GrB_NULL, GrB_NULL,
                                         MultipliesMultipliesSemiring<float>(),
                                         &r, &r, desc);
    reduce<float, float>(&error, GrB_NULL, PlusMonoid<float>(), &r_temp, desc);
    error = sqrt(error);
  }
}
```

```
template <class vertex> struct PR_F {
  double *p_curr, *p_next;
  vertex *V;
  PR_F(double *_p_curr, double *_p_next, vertex *_V)
      : p_curr(_p_curr), p_next(_p_next), V(_V) {}
  inline bool update(uintE s,
                     uintE d) {
    p_next[d] += p_curr[s] / V[s].getOutDegree();
    return 1;
  }
  inline bool updateAtomic(uintE s, uintE d) {
    writeAdd(&p_next[d], p_curr[s] / V[s].getOutDegree());
    return 1;
  }
  inline bool cond(intT d) { return cond_true(d); }
};
struct PR_Vertex_F {
  double damping;
  double addedConstant;
  double *p_curr;
  double *p_next;
  PR_Vertex_F(double *_p_curr, double *_p_next, double _damping, intE n)
      : p_curr(_p_curr), p_next(_p_next), damping(_damping),
        addedConstant((1 - _damping) * (1 / (double)n)) {}
  inline bool operator()(uintE i) {
    p_next[i] = damping * p_next[i] + addedConstant;
    return 1;
  }
};
struct PR_Vertex_Reset {
  double *p_curr;
  PR_Vertex_Reset(double *_p_curr) : p_curr(_p_curr) {}
  inline bool operator()(uintE i) {
    p_curr[i] = 0.0;
    return 1;
  }
};
template <class vertex> void Compute(graph<vertex> &GA, commandLine P) {
  long maxIters = P.getOptionLongValue("-maxiters", 100);
  const intE n = GA.n;
  const double damping = 0.85, epsilon = 0.0000001;
  double one_over_n = 1 / (double)n;
  double *p_curr = newA(double, n);
  { parallel_for(long i = 0; i < n; i++) p_curr[i] = one_over_n; }
  double *p_next = newA(double, n);
  { parallel_for(long i = 0; i < n; i++) p_next[i] = 0; }
  bool *frontier = newA(bool, n);
  { parallel_for(long i = 0; i < n; i++) frontier[i] = 1; }
  vertexSubset Frontier(n, n, frontier);
  long iter = 0;
  while (iter++ < maxIters) {
    edgeMap(GA, Frontier, PR_F<vertex>(p_curr, p_next, GA.V), 0, no_output);
    vertexMap(Frontier, PR_Vertex_F(p_curr, p_next, damping, n));
    {
      parallel_for(long i = 0; i < n; i++) {
        p_curr[i] = fabs(p_curr[i] - p_next[i]);
      }
    }
    double L1_norm = sequence::plusReduce(p_curr, n);
    if (L1_norm < epsilon)
      break;
    vertexMap(Frontier, PR_Vertex_Reset(p_curr));
    swap(p_curr, p_next);
  }
  Frontier.del();
  free(p_curr);
  free(p_next);
}
```

Algorithm 4. PR code for GraphBLAST (left), Ligra [76] (center right)

A.4 Connected components

```
void cc(Vector<int> *v, const Matrix<int> *A, int seed, Descriptor *desc) {
  Index A_nrows;
  CHECK(A->nrows(&A_nrows));
  Vector<bool> diff(A_nrows);
  Vector<int> parent(A_nrows);
  Vector<int> parent_temp(A_nrows);
  Vector<int> grandparent(A_nrows);
  Vector<int> grandparent_temp(A_nrows);
  Vector<int> min_neighbor_parent(A_nrows);
  Vector<int> min_neighbor_parent_temp(A_nrows);
  CHECK(parent.fillAscending(A_nrows));
  CHECK(min_neighbor_parent.dup(&parent));
  CHECK(min_neighbor_parent_temp.dup(&parent));
  CHECK(grandparent.dup(&parent));
  CHECK(grandparent_temp.dup(&parent));
  int succ = 0;
  for (int iter = 1; iter <= desc->descriptor_.max_niter_; ++iter) {
    CHECK(parent_temp.dup(&parent));
    mxv<int, int, int, int>(&min_neighbor_parent_temp, GrB_NULL, GrB_NULL,
                            MinimumSelectSecondSemiring<int>(), A, &grandparent,
                            desc);
    eWiseAdd<int, bool, int, int>(&min_neighbor_parent, GrB_NULL, GrB_NULL,
                                  MinimumSelectSecondSemiring<int>(),
                                  &min_neighbor_parent,
                                  &min_neighbor_parent_temp, desc);
    assignScatter<int, bool, int, int>(
        &parent, GrB_NULL, GrB_NULL, &min_neighbor_parent, &parent_temp, desc);
    eWiseAdd<int, bool, int, int>(&parent, GrB_NULL, GrB_NULL,
                                  MinimumPlusSemiring<int>(), &parent,
                                  &min_neighbor_parent, desc);
    eWiseAdd<int, bool, int, int>(&parent, GrB_NULL, GrB_NULL,
                                  MinimumPlusSemiring<int>(), &parent,
                                  &parent_temp, desc);
    extractGather<int, bool, int, int>(&grandparent, GrB_NULL, GrB_NULL,
                                       &parent, &parent, desc);
    eWiseMult<bool, bool, int, int>(&diff, GrB_NULL, GrB_NULL,
                                    MinimumNotEqualToSemiring<int, int, bool>(),
                                    &grandparent_temp, &grandparent, desc);
    reduce<int, bool>(&succ, GrB_NULL, PlusMonoid<int>(), &diff, desc);
    if (succ == 0)
      break;
    CHECK(grandparent_temp.dup(&grandparent));
    CHECK(desc->toggle(GrB_MASK));
    assign<int, bool, int, int>(&grandparent, &diff, GrB_NULL,
                                std::numeric_limits<int>::max(), GrB_ALL,
                                A_nrows, desc);
    CHECK(desc->toggle(GrB_MASK));
  }
  CHECK(v->dup(&parent));
}
```

```
struct CC_Shortcut {
  uintE *IDs;
  *prevIDs;
  CC_Shortcut(uintE *_IDs, uintE *_prevIDs) : IDs(_IDs), prevIDs(_prevIDs) {}
  inline bool operator()(uintE i) {
    uintE l = IDs[IDs[i]];
    if (IDs[i] != l)
      IDs[i] = l;
    if (prevIDs[i] != IDs[i]) {
      prevIDs[i] = IDs[i];
      return 1;
    } else
      return 0;
  }
};
struct CC_F {
  uintE *IDs;
  *prevIDs;
  CC_F(uintE *_IDs, uintE *_prevIDs) : IDs(_IDs), prevIDs(_prevIDs) {}
  inline bool update(uintE s, uintE d) {
    uintE origID = IDs[d];
    if (IDs[s] < origID) {
      IDs[d] = min(origID, IDs[s]);
    }
    return 1;
  }
  inline bool updateAtomic(uintE s, uintE d) {
    uintE origID = IDs[d];
    writeMin(&IDs[d], IDs[s]);
    return 1;
  }
  inline bool cond(uintE d) { return cond_true(d); }
};
template <class vertex> void Compute(graph<vertex> &GA, commandLine P) {
  long n = GA.n;
  m = GA.m;
  uintE *IDs = newA(uintE, n);
  *prevIDs = newA(uintE, n);
  {
    parallel_for(long i = 0; i < n; i++) {
      prevIDs[i] = i;
      IDs[i] = i;
    }
  }
  bool *all = newA(bool, n);
  { parallel_for(long i = 0; i < n; i++) all[i] = 1; }
  vertexSubset All(n, n, all);
  bool *active = newA(bool, n);
  { parallel_for(long i = 0; i < n; i++) active[i] = 1; }
  vertexSubset Active(n, n, active);
  while (!Active.isEmpty()) {
    edgeMap(GA, Active, CC_F(IDs, prevIDs), m / 20, no_output);
    vertexSubset output = vertexFilter(All, CC_Shortcut(IDs, prevIDs));
    Active.del();
    Active = output;
  }
  Active.del();
  All.del();
  free(IDs);
  free(prevIDs);
}
```

Algorithm 5. CC code for GraphBLAST (left), Ligra [76] (right)

A.5 Triangle counting

```
void tc(int *ntris, const Matrix<int> *A, Matrix<int> *B, Descriptor *desc) {
  Index A_nrows;
  CHECK(A->nrows(&A_nrows));
  CHECK(desc->toggle(graphblas::GrB_INP1));
  mxm<int, int, int, int>(B, A, GrB_NULL, PlusMultipliesSemiring<int>(), A, A,
                          desc);
  reduce<int, int>(ntris, GrB_NULL, PlusMonoid<int>(), B, desc);
}
```

```
template <class vertex>
long countCommon(vertex &A, vertex &B, uintE a, uintE b) {
  uintT i = 0, j = 0, nA = A.getOutDegree(), nB = B.getOutDegree();
  uintE *nghA = (uintE *)A.getOutNeighbors(),
        *nghB = (uintE *)B.getOutNeighbors();
  long ans = 0;
  while (i < nA && j < nB && nghA[i] < a &&
         nghB[j] < b) {
    if (nghA[i] == nghB[j])
      i++, j++, ans++;
    else if (nghA[i] < nghB[j])
      i++;
    else
      j++;
  }
  return ans;
}
template <class vertex> struct countF {
  vertex *V;
  long *counts;
  countF(vertex *_V, long *_counts) : V(_V), counts(_counts) {}
  inline bool update(uintE s, uintE d) {
    if (s > d)
      writeAdd(&counts[s], countCommon<vertex>(V[s], V[d], s, d));
    return 1;
  }
  inline bool updateAtomic(uintE s, uintE d) {
    if (s > d)
      writeAdd(&counts[s], countCommon<vertex>(V[s], V[d], s, d));
    return 1;
  }
  inline bool cond(uintE d) { return cond_true(d); }
};
struct intLT {
  bool operator()(uintT a, uintT b) { return a < b; };
};
template <class vertex>
struct initF {
  vertex *V;
  long *counts;
  initF(vertex *_V, long *_counts) : V(_V), counts(_counts) {}
  inline bool operator()(uintE i) {
    counts[i] = 0;
    quickSort(V[i].getOutNeighbors(), V[i].getOutDegree(), intLT());
    return 1;
  }
};
template <class vertex> void Compute(graph<vertex> &GA, commandLine P) {
  uintT n = GA.n;
  long *counts = newA(long, n);
  bool *frontier = newA(bool, n);
  { parallel_for(long i = 0; i < n; i++) frontier[i] = 1; }
  vertexSubset Frontier(n, n, frontier);
  vertexMap(Frontier, initF<vertex>(GA.V, counts));
  edgeMap(GA, Frontier, countF<vertex>(GA.V, counts), -1, no_output);
  long count = sequence::plusReduce(counts, n);
  cout << "triangle count = " << count << endl;
  Frontier.del();
  free(counts);
}
```

Algorithm 6. TC code for GraphBLAST (left), Ligra [76] (right)