UC Davis
Electrical & Computer Engineering

Title
High-Performance Linear Algebra-based Graph Framework on the GPU

Permalink
https://escholarship.org/uc/item/37j8j27d

Author
Yang, Carl Y

Publication Date
2019-05-31

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

High-Performance Linear Algebra-based Graph Framework on the GPU

By

CARL YUE YANG

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA, DAVIS

Approved:

Professor John D. Owens, Co-chair

Professor Aydın Buluç, Co-chair

Professor Chen-Nee Chuah

Committee in Charge

June 2019

Copyright © 2019 by
Carl Yue Yang
All rights reserved.

Abstract

High-Performance Linear Algebra-based Graph Framework on the GPU

By

Carl Yue Yang

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Davis

Professor John D. Owens, Co-chair
Professor Aydın Buluç, Co-chair

High-performance implementations of graph algorithms are challenging to implement on new parallel hardware such as GPUs, because of three challenges: (1) difficulty of coming up with graph building blocks, (2) load imbalance on parallel hardware, and (3) graph problems having low arithmetic ratio. To address these challenges, GraphBLAS is an innovative, on-going effort by the graph analytics community to propose building blocks based in sparse linear algebra, which will allow graph algorithms to be expressed in a performant, succinct, composable and portable manner. Initial research efforts in implementing GraphBLAS on GPUs has been promising, but performance still trails by an order of magnitude compared to state-of-the-art graph frameworks using the traditional graph-centric approach of describing operations on vertices or edges.

This dissertation examines the performance challenges of a linear algebra-based approach to building graph frameworks and describes new design principles for overcoming these bottlenecks. Among the new design principles is making *exploiting input sparsity* a first-class citizen in the framework. This is an especially important optimization, because it allows users to write graph algorithms without specifying certain implementation details thus permitting the software backend to choose the optimal implementation based on the input sparsity. *Exploiting output sparsity* allows users to tell the backend which values of the output in a single vectorized computation they do not want computed. We examine when it is profitable to exploit this output sparsity to reduce computational complexity. *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 the thesis have been implemented in GraphBLAST, an open-source high-performance graph framework on GPU developed as part of this dissertation. It is notable for being the first graph framework based in linear algebra to get comparable or faster performance compared to the traditional, vertex-centric backends. The benefits of design principles described in this thesis have been shown to be important for single GPU, and it will grow in importance when it serves as a building block for distributed implementation in the future and as a single GPU backend for higher-level languages such as Python. A graph framework based in linear algebra not only improves performance of existing graph algorithms, but in quickly prototyping new algorithms as well.

To my family

CONTENTS

List of Figures iv

List of Tables v

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Thesis Organization . . . 4

2 Background & Preliminaries 5
- 2.1 GPUs . . . 5
- 2.2 Sparse Matrix Formats . . . 5
- 2.3 Breadth-first-search . . . 6
- 2.4 Direction-optimized Breadth-first-search . . . 7
- 2.5 Notation . . . 9
- 2.6 Traversal is Matrix-vector Multiplication . . . 9

3 Related Work 11
- 3.1 Literature survey . . . 11
- 3.2 Previous systems . . . 12

4 Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU 14
- 4.1 Algorithms and Analysis . . . 14
- 4.2 Experiments and Results . . . 17
- 4.3 Conclusion . . . 20

5 Implementing Push-Pull Efficiently in GraphBLAS 21
- 5.1 Types of Matvec . . . 22
- 5.2 Relating Matvec and Push-Pull . . . 26
- 5.3 Optimizations . . . 28
- 5.4 Implementation . . . 32
- 5.5 Experimental Results . . . 36
- 5.6 Conclusion . . . 39

6 Design Principles for Sparse Matrix Multiplication on the GPU 40
- 6.1 Design Principles . . . 40
- 6.2 Parallelizations of CSR SpMM . . . 42
- 6.3 Experimental Results . . . 47
- 6.4 Conclusion . . . 50

7 Design of GraphBLAST 51
- 7.1 GraphBLAS Concepts . . . 52
- 7.2 Exploiting Input Sparsity (Direction-Optimization) . . . 60
- 7.3 Exploiting Output Sparsity (Masking) . . . 66

7.4 Load-balancing 67
7.5 Applications 71
7.6 Experimental Results 74

8 Conclusion **79**
8.1 Future Directions 79

References **83**

LIST OF FIGURES

1.1 Mismatch in existing frameworks 1

2.1 Matrix-graph duality 10

4.1 Workload distribution of three SpMSpV implementations 19
4.2 Performance comparison of three SpMSpV implementations 19

5.1 SpMV vs. SpMSpV 25
5.2 BFS traversal in linear algebra 27
5.3 Direction-optimized BFS 27
5.4 BFS traversal in detail 29
5.5 SpMV vs. SpMSpV applied to BFS 30
5.6 UML diagram of Vector interface 36
5.7 BFS performance comparison 38

6.1 Aspect ratio vs. performance 42
6.2 SpMV and SpMM load balance 43
6.3 SpMM tiling scheme 44
6.4 Aspect ratio vs. performance (this work) 48
6.5 SpMM performance comparison (selected) 48
6.6 SpMM performance comparison (all) 49

7.1 Decomposition of key GraphBLAS operations 56
7.2 BFS running example 57
7.3 Comparison of SpMV and SpMSpV. 61
7.4 Where this work on direction-optimization fits in literature. 63
7.5 Comparison with and without fused mask. 68
7.6 Graph algorithms using GraphBLAS 71
7.7 Performance comparison for GraphBLAST. 76
7.8 Hourglass design of GraphBLAST. 78

8.1 Scalability 80
8.2 Direction-optimization 81

List of Tables

4.1 BFS performance comparison 16
4.2 SpMSpV datasets 18
4.3 Workload distribution of three SpMSpV implementations 19
4.4 Scalability of three SpMSpV implementations 20

5.1 Matrix-vector computational complexity 24
5.2 BFS optimization summary 32
5.3 BFS datasets 37

6.1 ILP in SpMV and SpMM 45
6.2 SpMM datasets 47

7.1 GraphBLAST matrix and vector methods 53
7.2 GraphBLAST operations 54
7.3 GraphBLAST semirings and monoids 55
7.4 GraphBLAST descriptor settings 56
7.5 Applicability of design principles. 60
7.6 Matrix-vector complexity and sparsity 62
7.7 Direction-optimization switching criteria 65
7.8 GraphBLAST load-balancing 69
7.9 GraphBLAST datasets 74
7.10 GraphBLAST performance comparison 75
7.11 GraphBLAST lines of code 77

ACKNOWLEDGMENTS

Many people have contributed to making graduate career rewarding and enjoyable. First, I'd like to thank my PhD advisors John D. Owens and Aydın Buluç. John taught me about GPUs and that the right way to do computer science research is not to be satisfied with getting a good speed-up, but being able to explain why a speed-up exists. Aydin taught me about sparse linear algebra and pointed me to problems I was capable of solving. Chen-Nee Chuah, Zhaojun Bai and Venkatesh Akella formed the rest of my committee. Their insight and helpfulness improved the quality of this thesis.

I will be forever indebted to colleagues during my years at UC Davis. Yangzihao Wang, Yuechao Pan, Leyuan Wang, and Yuduo Wu have been great collaborators in the Gunrock project. Yangzihao taught me a lot about graph processing and shared my excitement in discovering commonalities between the vertex-centric and linear algebra-based perspectives. Yuechao provided a deep understanding about optimizing algorithms on the GPU. Saman Ashkiani, Jason Mak, Afton Geil, Muhammad Osama, Shari Yuan, Weitang Liu, Vehbi Bayraktar, Kerry Seitz, Collin Riffel, Andy Riffel, Shalini Venkataraman, Ahmed Mahmoud, Muhammad Awad, Yuxin Chen, Zhongyi Lin, and many others have also brightened up my life.

Thank you to all the people at the Department of Electrical and Computer Engineering at UC Davis, who helped me during my studies: Kyle Westbrook, Nancy Davis, Denise Christensen, Renee Kuehnau, Philip Young, Natalie Killeen, Fred Singh, Sacksith Ekkaphanh, and many more. They kept the department running smoothly and were always there for me.

I am thankful of the generosity of ideas and willingness to help amongst the graph research community. Marcin Zalewski and Peter Zhang taught me much about writing beautiful code, especially when I was getting started. Scott McMillan continually teaches me new ways of using C++. Tim Mattson and José Moreira taught me much about how to design interfaces.

Finally, I am also grateful for my family's support. It was with the help of Xinyan Xu that I was able to accomplish this. Her constant love and support make all this worthwhile.

Chapter 1

Introduction

Graphs are a representation that naturally emerges when solving problems in domains including bioinformatics [32], social network analysis [22], molecular synthesis [39], route planning [28]. Problem sizes can be number in over a billion vertices, so parallelization has become a must.

The past two decades has seen the rise of parallel processors to 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 developed under the DARPA SDH (Software Defined Hardware) program. Research into developing parallel hardware has succeeded in speeding up graph algorithms [61, 67]. 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 the programming language for parallel hardware (e.g. C++, CUDA, OpenMP, MPI).

To address this mismatch, many initiatives including NVIDIA's RAPIDS effort [59] 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 attractive open standard [18] that has been released for graph frameworks. It promises standard building blocks for graph algorithms in the language of linear algebra. This is exciting, because such a standard attempts to solve the following problems:

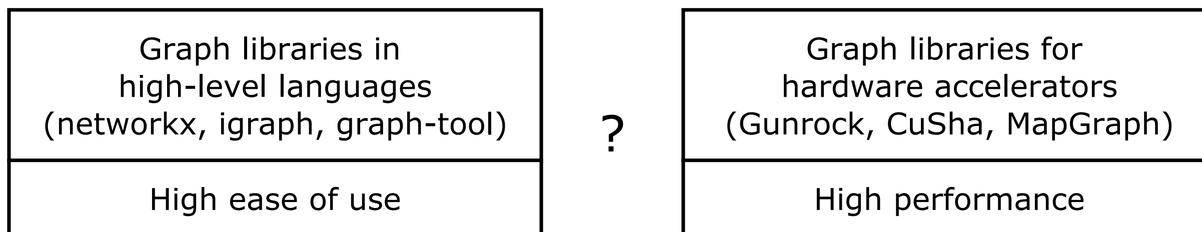


Figure 1.1: Mismatch between existing frameworks targeting high-level languages and hardware accelerators

1.1 Problem Statement

What is the right set of primitives for expressing graph algorithms? We will define the *right* set of primitives as one that fulfills the following goals:

1. *Performance portability*: Graph algorithm does not need modification to have high performance across hardware
2. *Concise expression*: Graph algorithms can be expressed in few lines of code
3. *High-performance*: Graph algorithms achieve state-of-the-art performance
4. *Scalability*: Framework is effective at small-scale and exascale

Firstly, the application code ought to require little to no change when targeting different backends. That is to say, the same application code ought to work just as well for single-threaded CPU as for GPU. Secondly, their application code ought to be concise. Thirdly, the graph primitives ought to have high-performance meeting that of "hardwired" code written in a low-level language targeting a particular hardware. Finally, graph algorithms written using the framework should be effective across a wide range of problem scales.

Goal 1 (*performance portability*) is central to the GraphBLAS philosophy, and it has made inroads in this regard with several implementations already being developed using this common interface [25, 53, 73]. Regarding Goal 2 (*concise expression*), GraphBLAS encourages users to think in a vectorized manner, which yields an order-of-magnitude reduction in SLOC as evidenced by Table 7.11. Before Goal 4 (*scalability*) can be achieved, Goal 3 *high-performance* on the small scale must first be demonstrated.

However, GraphBLAS has lacked high-performance implementations for GPUs. The GraphBLAS Template Library [73] 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?

Load imbalance Graph problems have irregular memory access pattern that makes it hard to extract parallelism from the data. 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 emphasizes making multiple memory accesses on the unstructured data instead of doing a lot of computations. Therefore, graph problems are often memory-bound rather than compute-bound. What can be done to reduce the *number of memory accesses*?

In other words, we are interested in answering the following question: What are the design principles required to build a GPU implementation based in 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 (graphics processing unit). Our implementation is for 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 CPU, 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 need to compare our framework against the state-of-the-art graph frameworks on the CPU and GPU, and hardwired GPU implementations, which are problem-specific GPU code that someone has hand-tuned for performance. The state-of-the-art graph frameworks we will be comparing against are Ligra [61] for CPU and Gunrock [67] for the GPU, which we will describe in detail in Section 3.2. The hardwired implementations will be Enterprise (BFS) [47], delta-stepping SSSP [24], pull-based PR [43], and bitmap-based triangle counting [14]. The array of graph algorithms we will be evaluating our system on are:

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

A description of these algorithms can be found in Section 7.5. We decided on these four applications, because based on a thorough literature survey by Beamer [9] they are considered the most common four applications across a variety of graph frameworks. Furthermore, they stress different facets of our framework. BFS and SSSP test how well we exploit *input sparsity*, PR tests sparse matrix-dense vector (SpMV) performance, and TC tests how well we exploit *output sparsity*.

Aside from these four algorithms, we have published elsewhere four algorithms built using our framework: Graph Projections, Seeded Graph Matching and Local Graph Clustering are applications built for the DARPA HIVE program [58], which is aimed at designing new graph processing hardware so our implementation on existing GPU hardware serves as a measure of goodness; Graph Coloring is a work where we compare our framework against Gunrock and hardwired implementations [57].

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

1.2 Thesis Organization

The rest of the thesis will be organized as follows: Chapter 2 gives background information on modern GPU architecture. Chapter 3 presents a survey of large-scale graph frameworks. Chapter 4 presents my work on exploiting *input sparsity* using column-based matrix multiplication to do breadth-first-search. Chapter 5 describes my work on exploiting *output sparsity* using row-based masked matrix multiplication and direction-optimized traversal to do breadth-first-search. Chapter 6 details my work on accelerating sparse matrix-dense matrix multiplication (SpMM). Chapter 7 leverages previous chapters to explain the design principles behind the architecture of GraphBLAST. Finally, Chapter 8 reviews future work that could be done in this area and the remaining research challenges to be solved.

Chapter 2

Background & Preliminaries

This section gives some background information on modern GPU architecture, sparse matrix formats, breadth-first-search, direction-optimized breadth-first-search, and introduces the duality between graph traversal and sparse matrix-vector multiplication that forms a cornerstone to our work.

2.1 GPUs

Modern GPUs are throughput-oriented manycore processors that rely on large-scale multi-threading to attain high computational throughput and hide memory access time. The latest generation of NVIDIA GPUs have up to 80 streaming multiprocessors (SMs), each with up to hundreds of arithmetic logic units (ALUs). GPU programs are called *kernels*, which run a large number of threads in parallel in a single-program, multiple-data (SPMD) fashion.

The underlying hardware runs an instruction on each SM on each clock cycle on a warp of 32 threads in lockstep. The largest parallel unit that can be synchronized within a GPU kernel is called a cooperative thread array (CTA), which is composed of warps. For problems that require irregular data access, a successful GPU implementation needs to (1) ensure coalesced memory access to external memory and efficiently use the memory hierarchy, (2) minimize thread divergence within a warp, and (3) maintain high occupancy, which is a measure of how many threads are available to run on the implementation on the GPU.

CPUs use branch prediction, speculative fetching, and large caches to *minimize* latency. By contrast, GPUs are throughput-oriented processors, instead relying on thread-level parallelism (TLP) to *hide* stalls. This means that for running certain irregular computations such as SpMV and SpMM, the bottleneck can be how long a multiply instruction immediately following a memory access must wait. While traditional analyses such as the roofline model [68] focus on compute-bound and memory-bound bottlenecks, we note GPU algorithms can also be latency-bound, which is when the GPU's parallelism is insufficient to hide the instruction latency.

2.2 Sparse Matrix Formats

An $m \times n$ matrix is often called *sparse* if its number of nonzeroes nnz is small enough compared to $\mathcal{O}(mn)$ such that it makes sense to take advantage of sparsity. The most straightforward

sparse matrix format is coordinate (COO) format. This format stores every nonzero as a triple $(i, j, \mathbf{A}_{ij})$. However, this format requires $3nnz$ memory for storage.

The compressed sparse row (CSR) format stores only the *column indices* and *values* of nonzeroes within a row. The start and end of each row is then stored in terms of the column indices and value in a *row offsets* (or row pointers) array. Hence, CSR only requires $m + 2nnz$ memory for storage.

Similarly to sparse matrix-dense vector multiplication (SpMV), a desire to achieve good performance on SpMM has inspired innovation in matrix storage formatting [2, 56]. These custom formats and encodings take advantage of the matrix structure and underlying machine architecture. Even only counting GPU processors, there exist more than sixty specialized SpMV algorithms and sparse matrix formats [31].

The vendor-shipped library cuSPARSE library provides two functions csrmm and csrmm2 for SpMM on CSR-format input matrices [54]. The former expects a column-major input dense matrix and generates column-major output, while the latter expects row-major input and generates column-major output. Among many efforts to define and characterize alternate matrix formats for SpMM are a variant of ELLPACK called ELLPACK-R [56] and a variant of Sliced ELLPACK called SELL-P [2]. However, there is a real cost to deviating from the standard CSR encoding. Firstly, the larger framework will need to convert from CSR to another format to run SpMM and convert back. This process may take longer than the SpMM operation itself. Secondly, the larger framework will need to reserve valuable memory to store multiple copies of the same matrix—one in CSR format, another in the format used for SpMM.

Ortega explores doing SpMM on a specialized matrix storage format, which is a variant on ELLPACK called ELLPACK-R [56]. Along with the usual two vectors that keep the nonzero index and value of standard ELLPACK, the -R variant keeps an additional vector that keeps the nonzero length of each row. Their insight is that by keeping the array in row-major format, they are able to obtain ILP for each thread through loop unrolling.

Anzt, Tomov and Dongarra use another matrix storage format that is a variant of Sliced ELLPACK called SELL-P to compute SpMM [2]. Their insight is that by forming row blocks, and padding ("P" stands for "padding") them with the number of threads assigned to each row, memory savings can be had over standard ELLPACK. At the same time, most of the advantages of ELLPACK over CSR are maintained.

2.3 Breadth-first-search

A common problem we are trying to solve is a breadth-first search on an unweighted directed or undirected graph $G = (V, E)$. V is the set of vertices of G, and E is the set of all ordered pairs (u, v), with $u, v \in V$ such that u and v are connected by an edge in G. A graph is undirected if for all $v, u \in V : (v, u) \in E \iff (u, v) \in E$. Otherwise, it is directed. For directed graphs, a vertex u is the child of another vertex v if $(v, u) \in E$ and the parent of another vertex v if $(u, v) \in E$.

Given a source vertex $s \in V$, a BFS is a full exploration of 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: *frontier* contains the vertices that are being explored at the current depth, *next* the vertices that can be reached from *frontier*, and *visited* the vertices reached so far.

Algorithm 1 Sequential breadth-first-search (BFS).

```
1:  procedure SEQUENTIALBFS(vertices, graph, source)
2:      frontier ← {source}
3:      next ← {}
4:      visited ← {-1, -1, ..., -1}
5:      depth ← 0
6:      while frontier ≠ {} do
7:          visited[i] ← depth ∀ i s.t. frontier[i] = 0
8:          for v ∈ frontier do
9:              for n ∈ neighbors[v] do
10:                 if visited[n] = -1 then
11:                     next ← next ∪ {n}
12:                 end if
13:             end for
14:         end for
15:         frontier ← next
16:         next ← {}
17:         depth ← depth + 1
18:     end while
19: return visited
20: end procedure
```

2.4 Direction-optimized Breadth-first-search

Push is the standard textbook way of thinking about BFS. At the start of each push step, each vertex in the *frontier* looks for its children and adds them to the *next* set if they have not been visited before. Once all children of the current frontier have been found, the discovered children are added to the visited array with the current depth, the depth is incremented, and the *next* set becomes the *frontier* of the next BFS step.

Pull is an alternative algorithmic formulation of BFS, yielding the same results but computing the *next* set in a different way. At the start of each pull step, each vertex in the *unvisited* set of vertices looks for its parents. If at least one parent is part of the *frontier*, we include the vertex in the *next* set.

Because either push or pull is a valid option to compute each step, we can achieve better overall BFS performance if we make the optimal algorithmic choice at each step. This is the key idea behind direction-optimized breadth-first-search (DOBFS), also known as push-pull BFS [10]. Push-pull can also be used for other traversal-based algorithms [13, 61]. DOBFS implementations use a heuristic function after each step to determine whether push or pull will be more efficient on the next step.

Algorithm 2 Direction-optimized BFS.

```
1:  procedure DIRECTIONOPTIMIZEDBFS(vertices, graph, source)
2:      frontier ← {source}
3:      next ← {}
4:      visited ← {-1, -1, ..., -1}
5:      depth ← 0
6:      while frontier ≠ {} do
7:          visited[i] ← depth ∀i s.t. frontier[i] = 0
8:          direction ← COMPUTEDIRECTION()
9:          if direction=PUSH then
10:             PUSHSTEP(vertices, graph, frontier, next, visited)
11:         else
12:             PULLSTEP(vertices, graph, frontier, next, visited)
13:         end if
14:         frontier ← next
15:         next ← {}
16:         depth ← depth + 1
17:     end while
18:     return visited
19: end procedure
```

Algorithm 3 Sequential push.

```
1: procedure PUSHSTEP(vertices, graph, frontier, next, visited)
2:     for v ∈ frontier do
3:         for n ∈ children[v] do
4:             if visited[n] = -1 then
5:                 next ← next ∪ {n}
6:             end if
7:         end for
8:     end for
9: end procedure
```

Algorithm 4 Sequential pull.

```
1:  procedure PULLSTEP(vertices, graph, frontier, next, visited)
2:      for v ∈ vertices do
3:          if visited[n] = -1 then
4:              for n ∈ parents[v] do
5:                  if n ∈ frontier then
6:                      next ← next ∪ {v}
7:                      break
8:                  end if
9:              end for
10:         end if
11:     end for
12: end procedure
```

2.5 Notation

At this point, we introduce some notation. We follow the MATLAB colon notation where $\mathbf{A}(:, i)$ denotes the ith column, $\mathbf{A}(i, :)$ denotes the ith row, and $\mathbf{A}(i, j)$ denotes the element at the (i, j)th position of matrix $\mathbf{A}$. We use $.*$ to denote the elementwise multiplication operator. For two frontiers $\mathbf{u}, \mathbf{v}$, their elementwise multiplication product $\mathbf{w} = \mathbf{u} .* \mathbf{v}$ is defined as $\mathbf{w}(i) = \mathbf{u}(i) * \mathbf{v}(i)\ \forall i$.

For a set of nodes $\mathbf{v}$, we will say the number of outgoing edges $nnz(m_{\mathbf{v}}^{+})$ is the sum of the number of outgoing edges of all nodes that belong to this set. Outgoing edges are denoted by a superscript '$+$', and incoming edges are denoted by a superscript '$-$'. That is, the number of incoming edges for a set of nodes $\mathbf{v}$ is

$$nnz(m_{\mathbf{v}}^{-}) = \sum_{i:\, \mathbf{v}(i) \neq 0} nnz(\mathbf{A}^T(i, :)). \tag{2.1}$$

For matrix $\mathbf{A}$, we will say the number of nonzero elements in it is $nnz(\mathbf{A})$. For a vector $\mathbf{v}$, we will say the number of elements in the vector is $nnz(\mathbf{v})$.

2.6 Traversal is Matrix-vector Multiplication

Since the start of graph theory, the duality between graphs and matrices has been established by the popular representation of a graph as an adjacency matrix [44]. After that time, it has become well-known that a vector-matrix multiply in which the matrix represents the adjacency matrix of a graph is equivalent to one iteration of breadth-first-search traversal. This is shown in Figure 2.1.

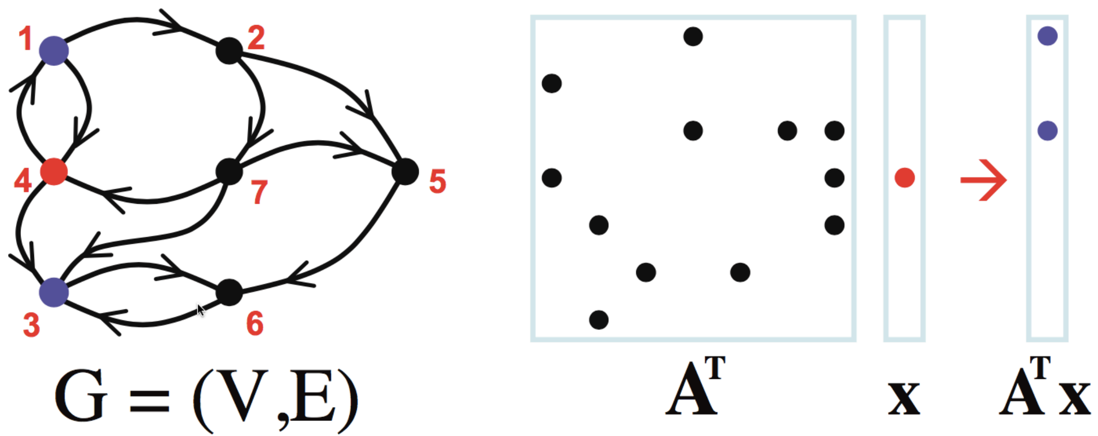


Figure 2.1: Matrix-graph duality. The adjacency matrix $\mathbf{A}$ is the dual of graph G. The current *frontier* (set of vertices we want to make a traversal from) is vertex 4. The next frontier is vertices 1 and 3, and is obtained by doing the matrix-vector multiplication. Therefore, the matrix-vector multiply (right) is the dual of the BFS graph traversal (left). Figure is based on Kepner and Gilbert's book [41].

Chapter 3

Related Work

Large-scale graph frameworks on multi-threaded CPUs, distributed memory CPU systems and massively parallel GPUs fall into three broad categories: vertex-centric, edge-centric and linear algebra-based.

3.1 Literature survey

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

3.1.1 Vertex-centric

Introduced by Pregel [48], vertex-centric frameworks are based on parallelizing by vertices. Vertex-centric frameworks follow an iterative convergent process (bulk synchronous programming model, or BSP) consisting of global synchronization barriers called *supersteps*. The computation in Pregel is inspired by distributed CPU programming model of MapReduce [27] and is based on message passing. At the beginning of the algorithm, all vertices are active. 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.

Its programming model is good for scalability and fault tolerance. However, standard graph algorithms in most Pregel-like graph processing systems suffer 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!.

3.1.2 Edge-centric (Gather-Apply-Scatter)

First introduced by PowerGraph [34], the edge-centric or Gather-Apply-Scatter (GAS) model is 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. In the linear algebraic model, edge-centric models are analogous to allocating to each processor an

even number of nonzeroes and computing matrix-vector multiply. For flexibility, PowerGraph also offers vertex-centric programming model, which is efficient on non-power law graphs.

3.1.3 Linear algebra-based

Linear algebra-based graph frameworks are pioneered by the Combinatorial BLAS (CombBLAS) [15], a distributed me mory 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 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 [62] is groundbreaking work. Traditionally, linear algebra-based frameworks have found difficulty gaining adoption, because they rely on users to understand how to express graph algorithms in terms of linear algebra. GraphMat addresses this problem by exposing a vertex-centric interface ot 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 [30] 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 [42], 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 [25] is notable for being the first GraphBLAS-compliant library. However, it currently only supports single-threaded CPU implementation.

3.2 Previous systems

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

3.2.1 Gunrock

Gunrock [67] 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. However, since Gunrock has many performance optimizations, Gunrock provides much flexibility in terms of choosing kernel variants the user wants to use. In our work, we aim to extract the performance Gunrock optimizations provide while delegating much of kernel selection work to the backend. This allows us to adhere to GraphBLAS's compact and easy to use user interface, while maintaining state-of-the-art performance.

3.2.2 Ligra

Ligra [61] is a CPU-based graph processing framework for shared memory. Its lightweight implementation is targeted at shared memory architectures and uses CilkPlus for its multi-

threading implementation. It is notable for being the first graph processing framework to generalize Beamer, Asanović and Patterson's direction-optimized BFS [10] 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 BLAS 3 operations (matrix-matrix multiplications).

Chapter 4

Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU

The motivating question for our work is given that the frontier vector (representing the set of vertices we would like to perform graph traversal from) is typically sparse, can we perform a sparse matrix-vector multiplication more efficiently than having to traverse the entire sparse matrix?

Before our work, there was research on sparse matrix-sparse vector in the CPU world [16, 33] and by the traversal matrix-vector duality discussed in Section 2.6 it was typical for traditional, graph-centric frameworks on the GPU. However, there were no sparse matrix-sparse vector implementations for the GPU, so we are the first to introduce this primitive to the GPU where the primitive is a cornerstone for any high-performance graph framework.

In this work, we show that a new primitive called sparse matrix-sparse vector multiplication (SpMSpV) is required in order to do graph algorithms efficiently. It performs favourably compared to sparse-matrix-dense vector multiplication (SpMV). Our contributions in this work are as follows:

1. We implement a promising algorithm for doing fast and efficient SpMSpV on the GPU.

2. We examine the various optimization strategies to solve the k-way merging problem that makes SpMSpV hard to implement on the GPU efficiently.

3. We provide a detailed experimental evaluation of the various strategies by comparing with SpMV and two state-of-the-art GPU implementations of breadth-first-search (BFS).

4.1 Algorithms and Analysis

Algorithm 5 gives the high-level pseudocode of our parallel algorithm. The sparse vector x is passed into the MULTIPLYBFS in dense representation. The STREAMCOMPACT consisting of

[1]This chapter substantially appeared as "Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU" [70], for which I was responsible for most of the research and writing.

Algorithm 5 SpMSpV multiplication algorithm for BFS.

Input: Sparse matrix G, sparse vector x (in dense representation)
Output: Sparse vector $w = G^T \times x$
1: **procedure** MULTIPLYBFS(G, x)
2: STREAMCOMPACT(x)
3: *ind* $\Leftarrow$ GATHER(G, x)
4: SORTKEYS(*ind*)
5: $w \Leftarrow$ SCATTER(*ind*)
6: **end procedure**

a scan and scatter is used to put the sparse vector into a sparse representation. The natively-supported scatter operation here is a moving of elements from the original array x into a list of new indices given by scan.

This sparse vector representation can be considered an analogue of the CSR format, with the simplification that since there is only one row, so the column-indices array C–which simplifies to the array with two elements $[0, m]$–will be replaced by a single variable, m.

Since the vector is sparse, we use something akin to outer product rather than SpMV's inner product. We do a linear combination on the rows of the matrix G. Even though the product we get is $G^T \times x$, we do not need to do a costly transposition in both memory storage and access since that is exactly the product we need for BFS. This way, we are only performing multiplication when we know for certain the resulting product is nonzero. This is the fundamental reason why SpMSpV is more work-efficient than SpMV.

To get the rows of G, we do a gather operation on the rows we are interested in and concatenate them into one array. The use of this single array is our attempt of solving the multiway merging problem in parallel, which is mentioned in Buluç and Madduri [16]. By concatenating into a single array, we are able to avoid atomic operations, which are known to be costly.

Going into more detail about this gather operation, we use the sparse vector x to get an index into graph G. (1) Then for all $i \in$ *ind* we gather from the graph's column-indices array obtaining two indices $C[i]$ and $C[i+1]$. These two indices give us the beginning and end of row i we are interested in. (2) Next, for all $h \in [C[i], C[i+1])$ we gather elements of row-offsets array $R[h]$ and call this set ind_i. The first two gather operations are shown as a single gather in Line 3 of Algorithm 5 and Algorithm 6.

In the case of Algorithm 6, we perform a third gather. This is to obtain the corresponding value $GVal[h]$ of node index $R[h]$ over the same interval $[C[i], C[i+1])$. We are now faced with the problem of doing a k-way merge of different-sized ind_i within *ind*. We tried three different approaches:

1. No sort.
2. Merge sort.
3. Radix sort.

We first try no sorting. Since the array is unsorted, adjacent threads do not write adjacent values; we instead scatter outputs to their memory destinations. The result is uncoalesced writes

Dataset	Runtime (ms)			Dataset Description			
	SpMSpV	Gunrock	b40c	Vertices	Edges	Max Degree	Diameter
ak2010	1.686	0.932	0.104	45K	25K	199	15
belgium_osm	63.937	13.053	1.277	1.4M	1.5M	9	630
coAuthorsDBLP	4.530	2.829	0.452	0.30M	0.98M	260	36
delaunay_13	1.085	0.820	0.117	8.2K	25K	10	142
delaunay_21	11.511	2.207	0.259	2.1M	6.3M	17	230
soc-LiveJournal1	73.722	33.953	21.117	4.8M	68.9M	20333	16
kron_g500-log21	70.935	15.194	23.423	2.1M	90M	131503	6

Table 4.1: Dataset descriptions and performance comparison of our SpMSpV implementation against two state-of-the-art BFS implementations on a single GPU for seven datasets.

into GPU memory, with a resulting loss of memory bandwidth. Davidson et al. [24] use a similar strategy when they remove duplicates in parallel in their single-source shortest-path (SSSP) algorithm.

Since we are skipping the sorting, we avoid the logarithmic time factor of merge sort mentioned by Buluç et al. [16]. We scatter 1's into a dense array using the concatenated array value as the index. This approach trades off less work in sorting for lower bandwidth from uncoalesced memory writes.

Algorithm 6 Generalized SpMSpV multiplication algorithm.

Input: Sparse matrix G, sparse vector x (in dense representation), operator $\oplus$, operator $\otimes$.
Output: Sparse vector $w = G^T \times x$.

```
 1: procedure MULTIPLY(G, x, ⊕, ⊗)
 2:     STREAMCOMPACT(x)
 3:     ind ⇐ GATHER(G, x)
 4:     GVal ⇐ GATHER(G, ind)
 5:     SORTPAIRS(ind, GVal)
 6:     for each j ∈ ind in parallel do
 7:         flag[j] ⇐ 1
 8:         val[j] ⇐ GVal[j] ⊗ x[j]
 9:         if ind[j] = ind[j − 1] then
10:             flag[j] ⇐ 0
11:         end if
12:     end for
13:     wVal ⇐ SEGREDUCE(val, flag, ⊕)
14:     w ⇐ SCATTER(wVal, ind)
15: end procedure
```

To increase our achieved memory bandwidth, we could perform the k-way merge by sorting. We first try a merge sort, which does $\mathcal{O}(f \log f)$ work, where f is the size of the frontier. Though this asymptotic complexity—which is $\mathcal{O}(m \log m)$ in the worst case—sounds bad com-

pared to the $\mathcal{O}(m)$ work of SpMV, it is actually much faster in practice due to the nature of BFS on typical graph topologies, which rarely visits a large fraction of the graph's vertices on a single iteration.

We also try radix sort, which has $\mathcal{O}(kf)$ work, where k is the length of the largest key in binary. We expect merge sort to be compute-bound; no-sorting to be memory-bound; and radix sort somewhere between the two. We investigate which is more efficient in practice.

Algorithm 6 is a generalized case of matrix multiplication parameterized by the two operations ($\oplus, \otimes$). If we set those two operations to ($\cup, \cap$), we obtain Algorithm 5. For low-diameter, power-law graphs, it is well-known that there are a few iterations when f becomes dense and these are the iterations that dominate the overall running time. For the remainder of BFS iterations, it is wasteful to use a dense vector.

We will investigate whether this crossing point is a fixed number independent of the total number of vertices or edges in the graph or whether it is determined by the percent of descendants f out of the total number of edges. The former would indicate a limit to SpMSpV's scalability since it would only be interesting for a small number of cases, while the latter would demonstrate that SpMSpV could outperform SpMV for BFS calculations on graphs of any scale provided they have a topology similar to those we perform our scalability tests.

4.2 Experiments and Results

We ran all experiments in this paper on a Linux workstation with 2× 3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, 528 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVIDIA's nvcc compiler (version 6.5.12). The C code was compiled using gcc 4.6.4. All results ignore transfer time (from disk-to-memory and CPU-to-GPU). The Gunrock code was executed using the command-line configuration `--src=0 --directed --idempotence --alpha=6`. The merge sort is from the Modern GPU library [8]. The radix sort is from the CUB library [50].

The datasets used in our experiments are shown in Table 4.1. The graph topology of the datasets varies from small-degree large-diameter to scale-free. The soc-LiveJournal1 (soc) and kron_g500-logn21 (kron) datasets are two scale-free graphs with diameter less than 20 and unevenly distributed node degree. The belgium-osm dataset has a large diameter with small and evenly distributed node degree.

Performance summary Looking at the comparison with two state-of-the-art BFS implementations, SpMSpV is between 2–4x slower. Nevertheless, this shows our implementation is a reasonable implementation, with runtime results in the same ballpark. With some Gunrock optimizations (that are not implemented in our system) turned off, the results are even closer.

One such BFS-specific optimization is direction-optimized traversal (discussed in detail in Chapter 5). This optimization is known to be effective when the frontier includes a substantial fraction of the total vertices [10]. Another reason may be kernel fusion [52]: b40c is careful to take advantage of producer-consumer locality by merging kernels together whenever possible. This way, costly reads and writes to and from global memory are minimized. Apart from that, both b40c and Gunrock use load-balancing workload mapping strategies during the neighbor list expanding phase of the traversal. Compared to b40c, Gunrock implements the direction-optimized traversal and more graph algorithms than BFS.

Dataset	Runtime (ms)		
	SpMSpV	SpMV	CPU
ak2010	1.686	0.427	0.00813
belgium_osm	63.937	97.280	0.0590
coAuthorsDBLP	4.530	6.213	5.507
delaunay_13	1.085	0.568	0.00571
delaunay_21	11.511	22.241	0.0128
soc-LiveJournal1	73.722	214.357	336.384
kron_g500-log21	70.935	230.609	753.737

Table 4.2: Performance comparison of our SpMSpV with SpMV for computing BFS on a single GPU for seven datasets.

Comparison with SpMV Table 2 compares SpMSpV's performance against SpMV. SpMSpV is 1.26x faster than SpMV at performing BFS on average. The primary reason is simply that SpMV does more work, performing multiplications on zeroes in the dense vector. The speed-up of SpMSpV is most prominent on scale-free graphs "soc" and "kron" where it is 2.9x and 3.3x faster. This is likely because on larger graphs, the work-efficiency of SpMSpV becomes prominent.

Such a conclusion is supported by the road network graph "belgium". It has a large number of edges, but both the average and max degrees are low while the diameter is high. In spite of being a graph of similar size to "delaunay_21", since not many edges need traversal every iteration there is not much difference in work-efficiency between the SpMSpV and SpMV. Perhaps superior load-balancing in the SpMV kernel is the difference maker. In the same vein, it can be seen that on the two smallest graphs "ak2010" and "delaunay_13", SpMV is 3.9x and 1.9x faster.

Figure 4.1 shows the impact of coalesced memory access on the scatter operation. Without sorting, scatter write takes up a majority of computation time for large datasets, but becomes neglible if prior sorting has been done. The only exception is for the road network graph belgium-osm, which has a high diameter and low node degree. This could be because the neighbor list is small every time and everything in the neighbor list is kept in sorted order, so there is little gained from performing a costly sort operation. The unnormalized data is given in Table 4.3.

Some parts of our SpMSpV implementations are common to all three of our approaches. We see some variance in this common code across our tests. Some of this variance is due to the method by which the execution times were measured, which was using the cudaEventRecord API. The rest of the variance is due to natural run-to-run variance of the GPU. This is why when possible, the runtimes taken were the average of ten iterations.

Figure 4.2 shows the runtime of BFS on a scale-free network ("kron") plotted against the number of edges traversed. SpMSpV implemented using radix sort and merge sort scale linearly, while SpMV (shown in Table 4.4 and SpMSpV with no sorting scale superlinearly. For a small number of edges, it is faster to do SpMSpV without sorting. Since SpMSpV seems to perform better than SpMV on bigger datasets, it seems that the answer as to whether the

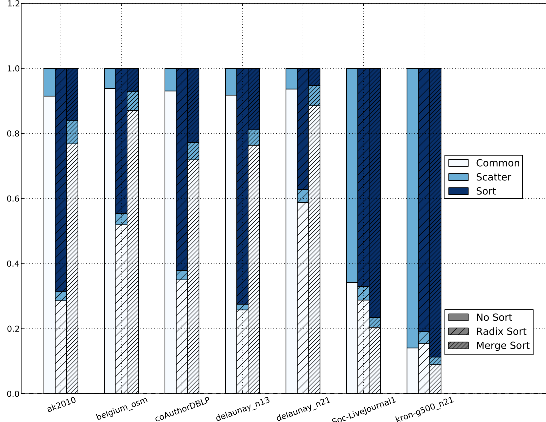


Figure 4.1: Workload distribution of three SpMSpV implementations. Shown are no sorting, radix sort and merge sort for the datasets listed in Table 1.

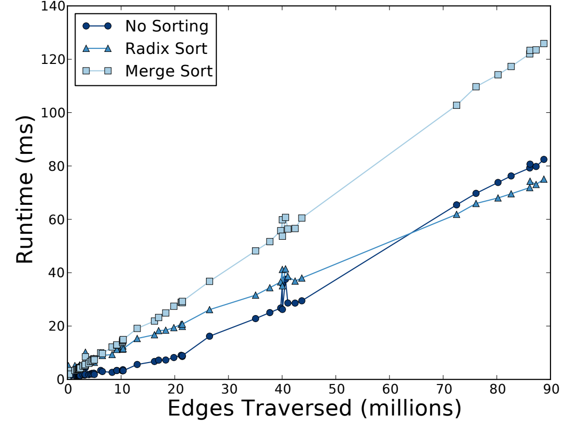


Figure 4.2: Performance comparison of three SpMSpV implementations on six differently-sized synthetically-generated Kronecker graphs with similar scale-free structure. The raw data used to generate this figure is given in Table 4.4. Each point represents a BFS kernel launch from a different node. Ten different starting nodes were used in this experiment.

	Runtime (ms)								
	No Sort			Radix Sort			Merge Sort		
Dataset	Common	Scatter	Sort	Common	Scatter	Sort	Common	Scatter	Sort
ak2010	0.5979	0.0556	0	0.5349	0.05433	1.2820	0.5387	0.0499	0.1128
belgium_osm	64.60	4.1906	0	63.03	4.1906	0	62.85	4.1843	5.1752
coAuthorsDBLP	5.4573	0.4067	0	5.3658	0.4403	9.5211	5.3508	0.3984	1.6931
delaunay_13	0.9395	0.0839	0	0.9146	0.0627	2.5720	0.9290	0.0573	0.2295
delaunay_21	11.18	0.7558	0	11.00	0.7479	6.9629	11.02	0.7506	0.6574
soc-LiveJournal1	21.71	41.80	0	21.67	3.1452	50.40	21.67	3.1452	50.40
kron-g500_n21	11.13	67.90	0	11.10	2.7280	58.24	11.12	2.7659	108.93

Table 4.3: Workload distribution of three SpMSpV implementations showing runtime (ms) on a single GPU for seven datasets. Common refers to time spent running the kernels common to all three implementations.

crossing point beyond which SpMV becomes more efficient than SpMSpV is governed not by a fixed frontier size, but rather as a function of both frontier size and the total number of edges as well. This indicates that SpMSpV is competitive with SpMV not just on datasets of limited size, but large datasets as well.

To explain the superlinear scaling, we offer a few likely explanations. One is that congestion degrades memory access latency [7]. As Figure 4.1 shows, the scatter writes are the difference between no sort and sort. One phenomenon that was observed was that if only a few iterations of merge and radix sort were performed, there would be no effect on scatter time and thereby increase the total execution time. Perhaps if a sorting algorithm that divides the array in a man-

Dataset	Runtime (ms)				Edge rate (MTEPS)			
	No Sorting	SpMV	Radix Sort	Merge Sort	No Sort	SpMV	Radix Sort	Merge Sort
kron-16	1.37	2.21	4.74	3.81	1401.9	868.3	405.07	503.9
kron-17	1.71	4.02	5.81	5.35	1923.5	819.6	567.3	615.2
kron-18	2.85	7.70	9.79	11.37	2764.8	1022.8	804.7	692.4
kron-19	6.79	19.94	16.85	22.31	2372.0	807.9	955.8	721.8
kron-20	21.08	75.97	29.25	43.13	1469.0	407.7	1058.7	718.2
kron-21	68.09	259.86	64.23	105.92	1087.7	285.0	1153.0	699.2

Table 4.4: Scalability of three SpMSpV implementations and one SpMV implementation (runtime and edges traversed per second) on a single GPU on six differently-sized synthetically-generated Kronecker graphs with similar scale-free structure. Radix sort and merge sort scale linearly; no sorting and SpMV show non-ideal scaling.

ner like quick sort or bucket sort were used, more coalesced memory access could be attained at the cost of additional computation.

Another way to express this idea is that there is an optimal compute to memory access ratio specific for each particular GPU hardware model. It is possible that the no sort implementation reached peak compute to memory access for dataset "kron_g500-logn18", but for larger datasets memory access grew faster than the amount of gather operations, so memory accesses were becoming degraded by congestion. The sorting methods may be closer to the compute-limited side of the compute to memory access peak, so the increased memory accesses are bringing them closer to peak performance.

4.3 Conclusion

In this paper we implement a promising algorithm for computing sparse matrix sparse vector multiplication on the GPU. Our results using SpMSpV show considerable performance improvement for BFS over the traditional SpMV method on power-law graphs. We also show that our implementation of SpMSpV is flexible and can be used as a building block for a linear algebra-based framework for implementing other graph algorithms.

An open research question now is how to optimize the compute to memory access ratio to maintain linear scaling. We showed merge sort and radix sort are good options, but it is possible a partial quick sort or a hybrid k-way merge algorithm such as the one presented by Leischner [46] can be used to obtain a better compute to memory access ratio, and better performance.

The SpMSpV algorithm used in this paper is generalizable to other graph algorithms through Algorithm 6. This algorithm is still being implemented in CUDA. By setting $(\oplus, \otimes)$ to $(+, \times)$, one performs standard matrix multiplication. A direction may be using SpMSpV as a building block for sparse matrix sparse matrix multiplication. Buluç and Gilbert's work in simulating parallel SpGEMM sequentially using SpMSpV has been promising [19]. Similarly, by setting $(\oplus, \otimes)$ to $(\min, +)$, one performs single-source shortest path (SSSP).

In this chapter, we saw that direction-optimized BFS is one reason Gunrock attained such high performance. In the next chapter, we address the problem of expressing direction-optimized BFS using linear algebra.

Chapter 5

Implementing Push-Pull Efficiently in GraphBLAS

In the previous chapter, we saw that direction-optimized BFS was a reason why our system performed worse than Gunrock. In this chapter, we solve the problem of how to express direction-optimized BFS using linear algebra.

In order to do so, we needed to factor Beamer's direction-optimized BFS [10] into 3 separable optimizations, and analyze them independently—both theoretically and empirically—to determine their contribution to the overall speed-up. This allows us to generalize these optimizations to other graph algorithms, as well as fit it neatly into a linear algebra-based graph framework. These 3 optimizations are, in increasing order of specificity:

1. Change of direction: Use *push* direction to take advantage of knowledge that the frontier is small, which we term *input sparsity*. When the frontier becomes large, go back to *pull* direction.

2. Masking: In *pull* direction, there is an asymptotic speed-up if we know *a priori* the subset of vertices to be updated, which we term *output sparsity*.

3. Early-exit: In *pull direction*, once a single parent has been found, the computation for that undiscovered node ought to exit early from the search.

Previous work by Beamer et al. [11] and Besta et al. [13] have observed that push and pull correspond to column- and row-based matrix-vector multiplication (Opt. 1). However, this knowledge is not exploited in the sole GraphBLAS implementation in existence so far, namely SuiteSparse GraphBLAS [25]. In SuiteSparse GraphBLAS, the BFS executes in only the forward (push) direction.

The key distinction between our work and that of Shun, Besta and Beamer is that while they take advantage of *input sparsity* using change of direction (Opt. 1), they do not analyze using *output sparsity* through masking (Opt. 2), which we show theoretically and empirically

[1] This chapter substantially appeared as "Implementing Push-Pull Efficiently in GraphBLAS" [72], for which I was the first author and responsible for most of the research and writing.

(in Table 5.1 and 5.2 respectively) is critical for high performance. Furthermore, we submit this speed-up extends to all algorithms for which there is *a priori* information regarding the sparsity pattern of the output such as triangle counting and enumeration [5], adaptive PageRank [40], batched betweenness centrality [17], maximal independent set [18], and convolutional neural networks [20].

Since the input vector can be either sparse or dense, we refrain from referring to this operation as SpMSpV (sparse matrix-sparse vector) or SpMV (sparse matrix-dense vector). Instead, we will refer to it as matvec (short for matrix-vector multiplication and known in GraphBLAS as GrB_mxv). Our contributions in this paper are:

1. We provide theoretical and empirical evidence of the asymptotic speed-up from masking, and show it is proportional to the fraction of nonzeroes in the expected output, which we term *output sparsity*.

2. We provide empirical evidence that masking is a key optimization required for BFS to attain state-of-the-art performance on GPUs.

3. We generalize the concept of masking to work on all algorithms where *output sparsity* is known before computation.

4. We show that direction-optimized BFS can be implemented in GraphBLAS with minimal change to the interface by virtue of an isomorphism between push-pull, and column- and row-based matvec.

5.1 Types of Matvec

The next sections will make a distinction between the different ways the matvec $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ can be computed. We define matvec as the multiplication of a sparse matrix with a vector on the right. This definition allows us to classify algorithms as row-based and column-based without ambiguity. We draw a distinction between SpMV (sparse matrix-dense vector multiplication) and SpMSpV (sparse matrix-sparse vector multiplication). Our analysis differs from previous work that focuses on the former, while we concentrate on the latter. Our novelty also comes from analysis of their masked variants, which is a mathematical formalism for taking advantage of *output sparsity* and to the best of our knowledge does not exist in the literature.

As mentioned in the introduction, we will henceforth refer to SpMV as row-based matvec, and SpMSpV as column-based matvec. We feel this is justified because although it is possible to implement SpMV in a column-based way and SpMSpV in a row-based way, it is generally more efficient to implement SpMV by iterating over rows of the matrix [65] and SpMSpV by fetching columns of the matrix $\mathbf{A}(:, i)$ for which $\mathbf{x}(i) \neq 0$ [4]. Here, we are talking about SpMV and SpMSpV without direct dependence on graph traversal. Hence we use the common, untransposed problem description $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ instead of that specific to graph traversal case.

5.1.1 Row- and column-based matvec

We wish to understand, from a matrix point of view, which of row- and column-based matvec is more efficient. We quantify efficiency with the random-access memory (RAM) model of computation. Since we assume the input vector must be read in both row- and column-based matvec, we will focus our attention on the number of random memory accesses into matrix $\mathbf{A}$.

Row-based matvec The efficiency of row-based matvec is straightforward. For all rows $i = 0, 1, ..., M$:

$$\mathbf{f}'(i) = \sum_{j:\, \mathbf{A}(i,j) \neq 0} \mathbf{A}(i, j) \times \mathbf{f}(j) \tag{5.1}$$

No matter what the sparsity of $\mathbf{f}$, each row must examine every nonzero, so the number of memory accesses into the matrix required to compute Equation 5.1 is simply $\mathcal{O}(nnz(\mathbf{A}))$.

Column-based matvec However, computing matvec ought to be more efficient if the vector $\mathbf{f}$ is all 0 except for just one element. We define such a situtation as *input sparsity*. Can we compute a result without touching all elements in the entire matrix? This is the benefit of column-based matvec: if only $\mathbf{f}(i)$ is nonzero, then $\mathbf{f}'$ is simply the ith column of $\mathbf{A}$ i.e., $\mathbf{A}(:, i) \times \mathbf{f}(i)$.

$$\mathbf{f}' = \sum_{i:\, \mathbf{f}(i) \neq 0} \mathbf{A}(:, i) \times \mathbf{f}(i) \tag{5.2}$$

When $\mathbf{f}$ has more than one non-zero element (when $nnz(\mathbf{f}) > 1$), we must access $nnz(\mathbf{f})$ columns in A. How do we combine these multiple columns into the final vector? The necessary operation is a multiway merge of $\mathbf{A}(:, i)\mathbf{f}(i)$ for all i where $\mathbf{f}(i) \neq 0$. Multiway merge (also known as k-way merge) is the problem of merging k sorted lists together such that the result is sorted [3]. It arises naturally in column-based matvec from the fact that the outgoing edges of a frontier do not form a set due to different nodes trying to claim the same child. Instead, one obtains $nnz(\mathbf{f})$ lists, and has to solve the problem of merging them together.

According to the literature, multiway merge takes $n \log k$ memory accesses where k is the number of lists and n is the length of all lists added together. For our problem where we have $k = nnz(\mathbf{f})$ and $n = nnz(m_{\mathbf{f}}^{+})$, so the multiway merge takes $\mathcal{O}(nnz(m_{\mathbf{f}}^{+}) \log nnz(\mathbf{f}))$.

Summary The complexity of row-based matvec is a constant; we need to touch every element of the matrix even if we want to multiply by a vector that is all 0's except for one index. On the other hand, the complexity of column-based matvec scales with $nnz(m_{\mathbf{f}}^{+})$. This matches our intuition, as well as the result of previous work [61], that shows column-based matvec should be more efficient when $\mathbf{f}$ is sparse.

5.1.2 Masked matvec

A useful variant of matvec is *masked matvec*. The intuition behind masked matvec is that it is a mathematical formalism for taking advantage of *output sparsity* (i.e., when we know which elements are zero in the output).

More formally, by masked matvec we mean computing $\mathbf{f}' = (\mathbf{A}\mathbf{f}) .* \mathbf{m}$ where vector $\mathbf{m} \in \mathbb{R}^{M \times 1}$ and $.*$ represents the element-wise multiplication operation. This concept of masking gives us two new definitions for row- and column-based masked matvec. By *row-based masked matvec*, we mean computing for all rows $i = 0, 1, ..., M$:

$$\mathbf{f}'(i) = \begin{cases} \sum_{j:\, \mathbf{A}(i,j) \neq 0} \mathbf{A}(i, j) \times \mathbf{f}(j) & \text{if } \mathbf{m}(i) \neq 0 \\ 0 & \text{if } \mathbf{m}(i) = 0 \end{cases} \tag{5.3}$$

Operation		Cost	Expected Cost
Row-based	unmasked	$\mathcal{O}(nnz(\mathbf{A}))$	$\mathcal{O}(dM)$
	masked	$\mathcal{O}(nnz(m^-_{\mathbf{m}}))$	$\mathcal{O}(d\, nnz(\mathbf{m}))$
Column-based	unmasked	$\mathcal{O}(nnz(m^+_{\mathbf{f}}) \log M)$	$\mathcal{O}(d\, nnz(\mathbf{f}) \log M)$
	masked	$\mathcal{O}(nnz(m^+_{\mathbf{f}}) \log M)$	$\mathcal{O}(d\, nnz(\mathbf{f}) \log M)$

Table 5.1: Four sparse matvec variants and their associated cost, measured in terms of number of memory accesses (actual and in expectation) into the sparse matrix $\mathbf{A}$ required.

Similarly for *column-based masked matvec*:

$$\mathbf{f}' = \mathbf{m}.* \sum_{i:\mathbf{f}(i)\neq 0} \mathbf{A}(:, i) \times \mathbf{f}(i) \tag{5.4}$$

The intuition behind masked matvec is that if more elements are masked out (i.e., $\mathbf{m}(i) = 0$ for many indices i), then we ought to be doing less work. Looking at the definition above, we no longer need to go through all nonzeroes in $\mathbf{A}$, but merely rows $\mathbf{A}(i, :)$ for which $\mathbf{m}(i) \neq 0$. Thus as shown in Figure 5.3c where $\mathbf{m} = \neg\mathbf{v}$, the number of memory accesses decreases to $\mathcal{O}(nnz(m^-_{\mathbf{m}}))$.

For column-based masked matvec, the number of memory accesses is that of computing column-based matvec, and doing an elementwise multiply with the mask, so the amount of computation does not decrease compared to the unmasked version. At this time, we do not know of an algorithm for column based matvec that can take advantage of the sparsity of $\mathbf{m}$ and thus reduce the number of memory accesses accordingly.

A summary of the complexity analysis above is shown in Table 5.1. We choose a matrix ('kron_g500-logn21' from the 10th DIMACS challenge [6]) and perform a microbenchmark to demonstrate the validity of this analysis. We will refer to it as 'kron' henceforth. We use the experimental setup described in Section 5.5. We measure the runtime of four variants given above for increasing frontier sizes (for the two column-based matvecs), and increasing unvisited node counts (for the two row-based matvecs):

1. Row-based: increase $nnz(\mathbf{f})$, no mask
2. Row-based masked: $nnz(\mathbf{f}) = M$, increase $nnz(\mathbf{m})$
3. Col-based: increase $nnz(\mathbf{f})$, no mask
4. Col-based masked: increase $nnz(\mathbf{f})$, increase mask at $\frac{2}{3} nnz(\mathbf{f})$

Nodes were selected randomly to belong to the frontier and unvisited nodes. Here, we are using frontier size $nnz(\mathbf{f})$ as a proxy for $nnz(m_{\mathbf{f}})$. The number of outgoing edges $nnz(m_{\mathbf{f}}) \approx d\, nnz(\mathbf{f})$, where d is the average number of outgoing edges per node. Similarly, we use $nnz(\mathbf{m})$ as a proxy for $nnz(m_{\mathbf{m}})$.

The results are shown in Figure 5.1. They agree with our derivations above. For a given matrix, the row-based matvec's runtime is independent of a varying frontier size and unvisited

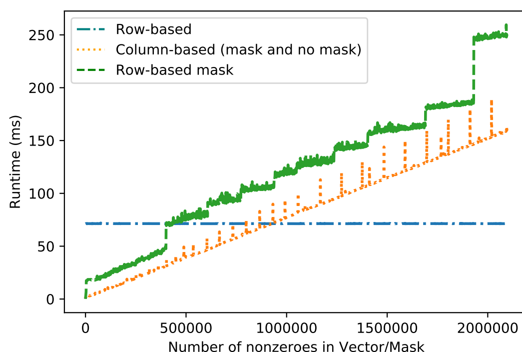


Figure 5.1: Runtime in milliseconds for row-based and column-based matvec in their masked and unmasked variants for matrix 'kron' as a function of $nnz(\mathbf{f})$ and $nnz(\mathbf{m})$.

node count. The runtime of the column-based matvec and the masked row-based matvec both increase with frontier size and unvisited node count, respectively. For low values of either frontier size or unvisited node count, doing either column-based matvec or masked row-based matvec is more efficient than row-based matvec. For high values of either frontier size or unvisited node count, doing the row-based matvec can be more efficient.

In Section 5.3, we will show that it is by staying in this region (low frontier size and low unvisited node count) through intelligent switching between column-based and row-based masked matvecs is what enables an entire BFS traversal to complete in less time than even a single row-based matvec.

5.1.3 Structural complement

Another useful concept is the *structural complement*. Recall the intuition behind masked matvec is that if the mask vector $\mathbf{m}$ is 1 at some index i, then it will allow the result of the computation to be passed through to the output $\mathbf{f}'(i)$. The structural complement operator $\neg$ is a user-controlled switch that lets them invert this rule: all the indices i for which $\mathbf{m}$ were 1 will now prevent the result of the computation to be passed through to the output $\mathbf{f}'(i)$, while the indices that were 0 will allow the result ot be passed through.

5.1.4 Generalized semirings

One important feature that GraphBLAS provides is that it allows users to express different traversal graph algorithms such as BFS, SSSP (Bellman-Ford), PageRank, maximal independent set, etc. using matvec and matmul [41]. This way, the user can succinctly express the desired graph algorithm in a way that makes parallelization easy. This is analogous to the key role Level 3 BLAS (Basic Linear Algebra Subroutines) plays in scientific computing; it is much easier to optimize for a set of standard operations than have scientists optimize every application all the way down to the hardware-level. The mechanism in which they are able to do so is called *generalized semirings*.

What generalized semirings do is allow the user to replace the standard matrix multiplication and addition operation over the real number field with zero-element 0 $(\mathbb{R}, \times, +, 0)$ by any operation they want over arbitrary field $\mathbb{D}$ with zero-element $\mathbb{I}$ $(\mathbb{D}, \otimes, \oplus, \mathbb{I})$. We refer to

the latter as *matvec over semiring* $(\mathbb{D}, \otimes, \oplus, \mathbb{I})$. We also have the row-based and column-based equivalents for all semirings. For example, *row-based matvec over semiring* $(\mathbb{D}, \otimes, \oplus, \mathbb{I})$ is:

$$\mathbf{f}'(i) = \bigoplus_{\substack{\mathbf{A}(i,j) \neq \mathbb{I} \\ j=0}}^{n} \mathbf{A}(i,j) \otimes f(j)$$

For row-based masked and column-based masked matvec over semirings, we generalize the element-wise operation to be $\odot\colon \mathbb{D} \times \mathbb{D}_2 \rightarrow \mathbb{D}$ where $\mathbb{D}_2$ is the set of allowable values of the mask vector $\mathbf{m}$ and $\mathbb{D}$ is the set of allowable values of the matrix $\mathbf{A}$ and vector $\mathbf{f}$. For example, *row-based masked matvec over semiring* $(\mathbb{D}, \otimes, \oplus, \mathbb{I})$ *and element-wise multiply* $\odot\colon \mathbb{D} \times \mathbb{D}_2 \rightarrow \mathbb{D}$ is:

$$\mathbf{f}'(i) = \mathbf{m}. \odot \bigoplus_{\substack{\mathbf{A}(i,j) \neq \mathbb{I} \\ j=0}}^{n} \mathbf{A}(i,j) \otimes f(j)$$

As an example, if the user wants to change from BFS to SSSP, they must specify a change to the semiring from the Boolean semiring $(\{0, 1\}, OR, AND, 0)$ to the tropical semiring $(\mathbb{R} \cup \{\infty\}, min, +, \infty)$ and removing the masks from the GrB_assign and GrB_mxv. Then with this simple change to 2 lines of code, the GraphBLAS application code would support high-performance SSSP code on any hardware backend for which there exists a GraphBLAS implementation.

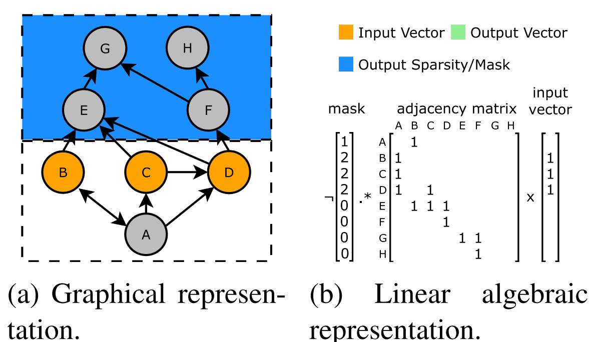
5.2 Relating Matvec and Push-Pull

In this section, we discuss the connection between masked matvec and the three optimizations inherent to DOBFS. Then, we discuss two closely related optimizations that were not in the initial direction-optimization paper [10] by Beamer, Asanović, and Patterson. In recent work, these authors looked at matvec in their row- and column-based variants for PageRank [11]. They examine three blocking methods (cache, propagation and deterministic propagation) for computing matvec using row- and column-based approaches. Besta et al. also observed the duality between push-pull and row- and column-based matvec in the context of several graph algorithms. They give a theoretical analysis on three parallel random access memory (PRAM) variants for differences between push-pull. We extend their push-pull analysis to include the concept of masking, which is needed to take advantage of output sparsity and express early exit.

5.2.1 Connection with push

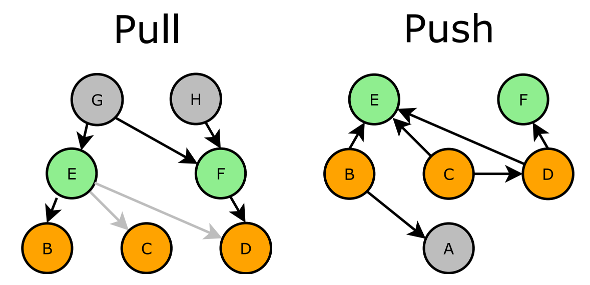
To demonstrate the connection with push-pull, we consider the formulation of the problem using $\mathbf{f}' = \mathbf{A^T}\mathbf{f}. * \neg\mathbf{v}$ in the specific context of one BFS iteration. In graphical terms, this is visualized as Figure 5.2a. Our current frontier is shown by nodes marked in orange. The visited vector $\mathbf{v}$ indicates the already visited nodes A, B, C, D.

We will first consider the push case as shown in Figure 5.2d. We must examine all edges leaving the current frontier. Doing so, we examine the children of B, C, D, and combine them using a logical OR. This allows us discover nodes A, E, F. From these 3 nodes, we must filter using the visited vector $\mathbf{v}$ and eliminate A from our frontier. This leaves us with the two nodes



(a) Graphical representation.

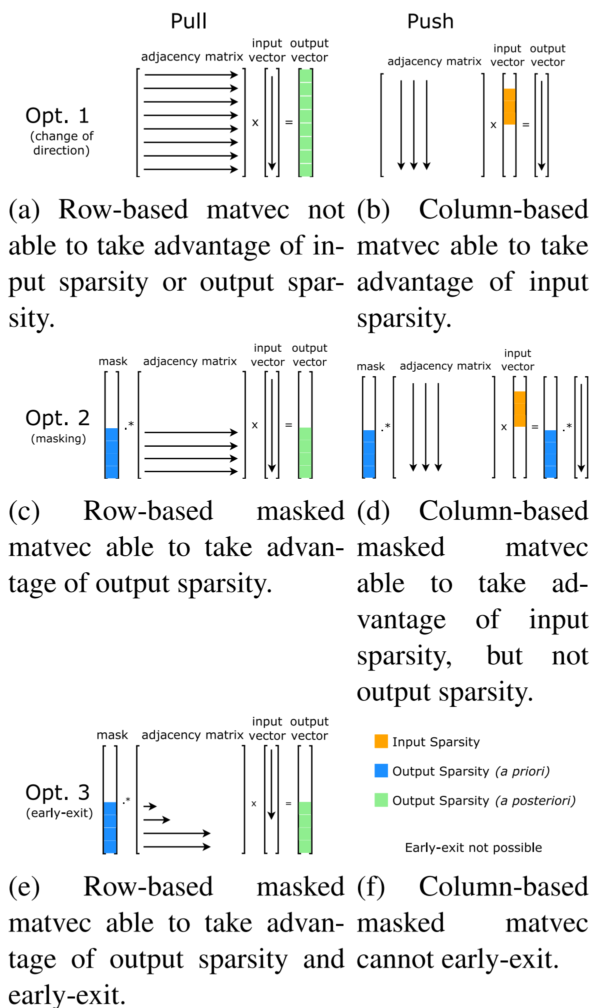
(b) Linear algebraic representation.



(c) Pull iteration: Start from unvisited vertices (in gray and green), then find their parents. Gray edges indicate ones that need not be checked due to early exit.

(d) Push iteration: Start from frontier (in orange), then find their children (in green).

Figure 5.2: Simple example showing BFS traversal from the 3 nodes marked orange. There is a one-to-one correspondence between the graphical representation of both traversal strategies and their respective matvec equivalents in Figure 5.3.



(a) Row-based matvec not able to take advantage of input sparsity or output sparsity.

(b) Column-based matvec able to take advantage of input sparsity.

(c) Row-based masked matvec able to take advantage of output sparsity.

(d) Column-based masked matvec able to take advantage of input sparsity, but not output sparsity.

(e) Row-based masked matvec able to take advantage of output sparsity and early-exit.

(f) Column-based masked matvec and cannot early-exit.

Figure 5.3: The three optimizations known as "direction-optimized" BFS. We are the first to generalize Optimization 2 by showing that masking can achieve asymptotic speedup over standard row-based matvec when output sparsity is known before computation (i.e., *a priori*).

marked in green E, F as the newly discovered nodes. In matvec terms, our operation is the same: we find the neighbors of the current frontier (represented by columns of $\mathbf{A^T}$) and merge them together before filtering using $\mathbf{v}$. This is a well-known result [16, 33].

5.2.2 Connection with pull

Now let us consider the pull case shown in Figure 5.2c. Here, the traversal pattern is different, because we must take the unvisited vertices $\neg\mathbf{v}$ as our starting point (Opt. 2: masking). We start from each unvisited vertex E, F, G, H and look at each node's parents. Once a single parent has been found to belong in the visited vector, we can mark the node as discovered (Opt. 3:

early-exit). In matvec terms, we apply the unvisited vector $\mathbf{v}$ as a mask to our matrix to take advantage of *output sparsity*. Since we know that the first four nodes with values $(0, 1, 1, 1)$ will be filtered out anyways, we can skip computing matvec for them. For the rest, we will begin examining each unvisited node's parents until we find one that is in the frontier. Once we have found one, this is sufficient to break the loop and early-exit.

In mathematical terms, performing the early-exit is justified inside an matvec inner loop as long as the addition operation of the matvec semiring is an OR that evaluates to true. This is the same principle by which C compilers allow short-circuit evaluation. This can easily be implemented in the GraphBLAS underlying implementation by adding an if-statement that checks whether the matvec semiring is logical OR.

Algorithm 7 BFS using Boolean semiring $(\{0, 1\}, OR, AND, 0)$ with equivalent GraphBLAS operations highlighted. For GrB_mxv, the operations are changed from their standard matrix multiplication meaning to become $\times = AND, + = OR$. GrB_assign uses the standard matrix multiplication meanings for the $\times$ and $+$.

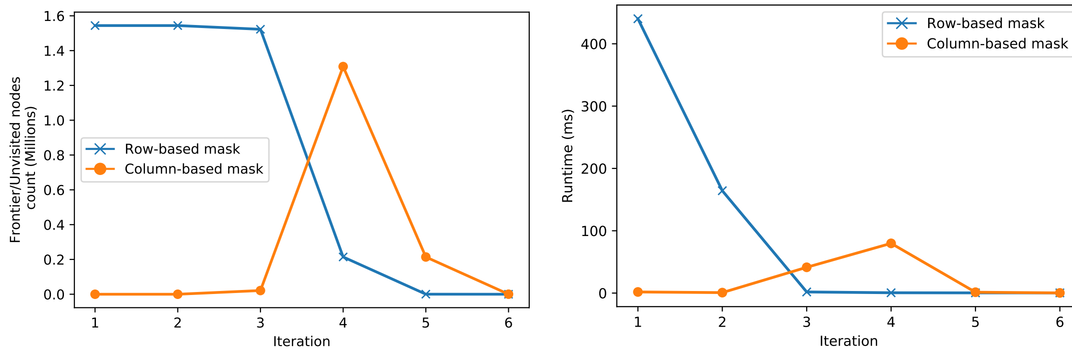
```
 1: procedure GRB_BFS(Vector v, Graph A, Source s)
 2:     Initialize d ← 1
 3:     Initialize f(i) ← { 1, if i = s ; 0, if i ≠ s }       ▷ GrB_Vector_new
 4:     Initialize v ← [0, 0, ..., 0]                          ▷ GrB_Vector_new
 5:     Initialize c ← 1
 6:     while c > 0 do
 7:         Update v ← f × d + v                               ▷ GrB_assign
 8:         Update f ← A^T f .* ¬v                             ▷ GrB_mxv
 9:         Compute c ← Σ_{i=0}^{n} f(i)                        ▷ GrB_reduce
10:         Update d ← d + 1
11:     end while
12: end procedure
```

What we propose is that the pull case can be expressed by the same formula $\mathbf{f}' = \mathbf{A}^\mathbf{T}\mathbf{f} .* \neg\mathbf{v}$ as the one in the GraphBLAS C API [18]. The full algorithm is shown in Algorithm 7. This will allow the GraphBLAS backend to take the function call requested by the user and make a runtime decision as to whether to use the column-based matvec or the row-based matvec (Opt. 1: change of direction).

5.3 Optimizations

In this section, we discuss in-depth the five optimizations mentioned in the previous section. We also analyze their suitability for generalization to speeding up matvec for other applications.

1. Change of direction
2. Masking
3. Early-exit



(a) Frontier count and unvisited node count.

(b) Push and pull runtime.

Figure 5.4: Breakdown of edge types in frontier during BFS traversal of Kronecker scale-21 graph (2M vertices, 182M edges).

4. Operand reuse

5. Structure only

Opt. 1, 2 and 3 form the commonly used definition of DOBFS from the paper that discovered it [10]. Opt. 4 and 5 are also key to a performant BFS. The impact of these five optimizations are summarized in Table 5.2.

5.3.1 Optimization 1: Change of direction

When the frontier becomes large, instead of each frontier node looking for its children and adding them to the next frontier (*push*), it becomes more efficient if each *unvisited* node looks for its parents (*pull*). Near the end of the computation, the number of frontier edges once again falls, and it is profitable once more to return to *push*. Efficient DOBFS traversals on scale-free graphs result in three distinct phases:

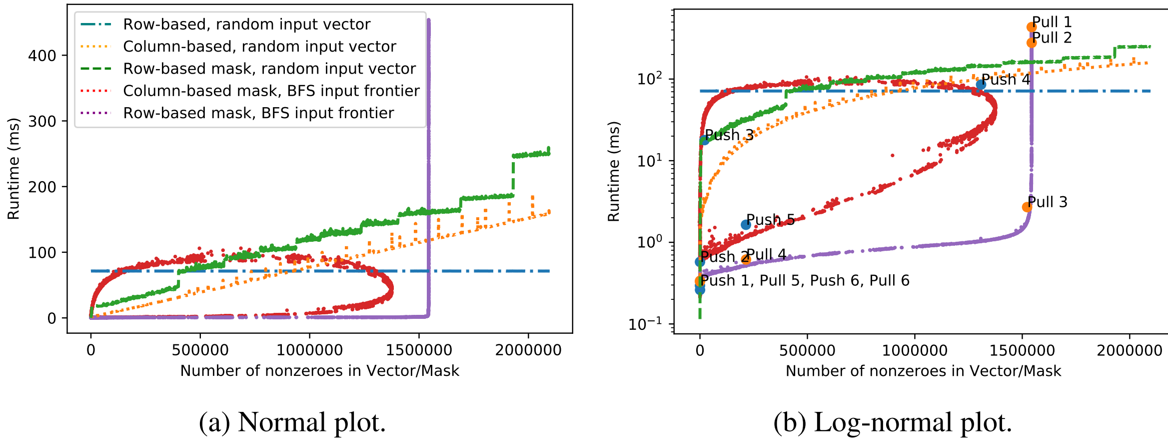
1. Push phase: Frontier is small, unvisited vertices is large.

2. Pull phase: Frontier is medium, unvisited vertices is large.

3. Push phase: Frontier is small, unvisited vertices is small.

Figure 5.4 shows an empirical representation of this phenomenon on a Kronecker graph of scale-21 with 2M vertices and 182M edges. In Iterations 1–2 of the BFS, the frontier size is small. Similarly, the number of unvisited vertices is big, so it is profitable to use *push*. In Iteration 6, the frontier size falls once more, so it is worthwhile to go back to *push*. The frontier size and number of unvisited vertices is comparable for Iterations 2 and 3. However, performance of row-based with mask and column-based with mask is drastically different between these two iterations. The row-based with mask runtime drops precipitously, but the column-based with mask runtime increases.

To solve this problem, we perform another microbenchmark that differs in two respects compared to Figure 5.1. First, in the previous benchmark, we generated random vectors as



(a) Normal plot.

(b) Log-normal plot.

Figure 5.5: Runtime in milliseconds for row-based and column-based matvec in their masked and unmasked variants for matrix 'kron' as a function of $nnz(\mathbf{f})$ and $nnz(\mathbf{m})$. Input vectors and masks are generated using 2 methods: (1) random input vectors, (2) based on sampling BFS iterations from 1000 random sources on graph 'kron_g500-logn21'. Push 1 means Iteration 1 of the push-only BFS. Pull 1 means Iteration 1 of the pull-only BFS. For this graph, 2 iterations of push followed by 3 of pull, then 1 iteration of push or pull, yields the best performance.

input and mask; here we launch BFS from 1000 different sources and plot the per-iteration runtime as a function of input frontier size (in the case of column-based) and unvisited nodes (in the case of row-based with mask), so the vectors have semantic meaning. Second, for row-based with mask we activate the early exit optimization. The result of the microbenchmark is shown in Figure 5.5. Interestingly, the runtimes of the row-based and column-based matvecs look very different depending on whether the input vector is random or whether it represents a BFS frontier.

We begin by analyzing the column-based-with-mask case (the red oval in Figure 5.5). This interesting shape is characteristic of power-law graphs (of which 'kron' is a member) that have a small number of nodes with many edges (supernodes) but most nodes with only a few. We examined a few examples of the runtime data. Column-based-on-graph-data (push-based BFS) progresses with increasing iteration count in a clockwise-direction. In the first phase of the BFS, the algorithm quickly discovers a few supernodes, which dramatically increases its runtime. The BFS then reaches a peak in the frontier size (Iteration 4 of Figure 5.4a), at which point the frontier begins to fall. Its return to low frontier size corresponds to the bottom of the oval. Despite a comparable nonzero count in the input vector, this phase is notable for its lack of supernodes, which keeps runtime at a minimum.

Row-based mask has a different pattern. There, the pull-based BFS begins at the top of the backwards 'L', then moves down and towards the left with increasing iteration count. When there have only been one or two nodes visited, we most likely have not discovered a super node yet, so the runtime is high (around 1.5M in Figure 5.5). We found it took on average 79 parent checks before a valid parent was found. After the first 20k or 30k nodes have been visited, it is with high probability that a supernode has been visited. In either case, we found that after 2 BFS iterations, the average numbers of parents examined before a valid parent was found

dropped from 79 to 1.3. This concept of early-exit does not exist in the matvec of Figure 5.1, so there we get a different result. The line peaks around 1.5M, because that is the size of the largest connected component.

In light of DOBFS, it becomes clear from looking at Figure 5.5b that at the start of the BFS, the push BFS is below the blue line around $(0, 10^{-1})$, but the pull BFS is above the blue line, so it is more efficient to do push BFS for the first few iterations. By Iteration 3, the push-based has increased to above the blue line, while the pull-based has dropped sharply below. At this point, it is more efficient to do pull BFS. Near the end of the algorithm, both algorithms continue to improve in efficiency, so either algorithm will suffice.

5.3.2 Optimization 2: Masking

As described in Section 5.1, masking means computing only the rows whose value we know *a priori* must be updated. In Figure 5.3c, this means depending on the unvisited nodes (mask), we can perform a matvec using just the latter four rows of the matrix. This yields the algorithmic speed-up of $\mathcal{O}(nnz(m_{\mathbf{m}}^{-}))$ over $\mathcal{O}(nnz(\mathbf{A}))$.

5.3.3 Optimization 3: Early-exit

In the pull phase, an undiscovered node searches for a parent in the frontier. Once such a parent has been found, further computation for that undiscovered node is unnecessary and can potentially halt. For the nodes who do not have parents that have been previously visited, early-exit has no benefit. According to Table 5.2, this optimization yielded the greatest speed-up. This optimization is only generalizable to semirings with Boolean operators that support short-circuiting.

5.3.4 Optimization 4: Operand reuse

Since the set of the visited node list is always a superset of the frontier node list, we can simply use the visited node list in place of the frontier. Gunrock [67] notes that $\mathbf{f} \subset \mathbf{v}$ and computes $\mathbf{A^T v} .* \neg\mathbf{v}$ instead of $\mathbf{A^T f} .* \neg\mathbf{v}$. This is a powerful optimization, because computing the latter means that during the iteration in which we are switching from push to pull; we get the costly sparse to dense frontier vector conversion for free because in the above expression, the frontier $\mathbf{f}$ is not required as part of the input.

5.3.5 Optimization 5: Structure only

The matrix values and sparse vector values do not need to be accessed for BFS. This optimization takes advantage of the fact for the purposes of a BFS, the matrix can be implicitly treated as a Boolean matrix, because we treat the existence of sparse matrix column indices as a Boolean 1, and non-existence as Boolean 0. The majority of the speed-up comes during the multiway merge. In Section 5.4, we say that we implement this multiway merge using a radix sort. This radix sort is often the bottleneck of the algorithm, so if we use this optimization, we are reducing a key-pair sort to a key-only sort, which will reduce the number of memory accesses by a factor of 2.

Optimization	Performance (GTEPS)	Speed-up
Baseline	0.874	—
Structure only	1.411	1.62×
Direction-optimization	1.527	1.08×
Masking	3.932	2.58×
Early exit	15.83	4.02×
Operand reuse	42.44	2.68×

Table 5.2: Impact of the four optimizations described in this section on the performance measured in billions of traversed edges per second on 'kron_g500-logn21'. These optimizations are cumulative, meaning the next optimization is stacked on top of the previous one. Speedups are standalone.

5.3.6 Generality

Change of direction can be generalized to other algorithms including betweenness centrality, personalized PageRank, and SSSP, with similar tradeoffs between row-based and column-based approaches (Table 5.1). In SSSP, for example, despite the workfront evolution (how the frontier changes between iterations) being completely different from Figure 5.4a, a simple 2-phase direction-optimized traversal can be used where the traversal is begun using unmasked column-based matvec, and switches to row-based matvec when the frontier becomes large enough that row-based is more efficient.

Masking can be generalized to any algorithms where the *output sparsity* is known ahead of computation. This includes algorithms such as triangle counting and enumeration [5], adaptive PageRank [40], batched betweenness centrality [17], maximal independent set [18], and convolutional neural networks [20]. In all of these algorithms, an optimal algorithm will want to use the knowledge that some output elements will be zero (e.g., when the PageRank value has converged for a particular node). In these cases, our proposed elementwise multiply formalism provides the mathematical theory to take advantage of this *output sparsity* yielding an asymptotic speed-up of $\mathcal{O}(\frac{dM}{dnnz(\mathbf{m})}) = \mathcal{O}(\frac{M}{nnz(\mathbf{m})})$.

Operand reuse is generalizable to any traversal-based algorithm for which computing $\mathbf{A^T v}$ in place of $\mathbf{A^T f}$ gives the correct result. We give SSSP and personalized PageRank as examples for which this holds true. However, early-exit and structure only are only generalizable to semirings that operate on Booleans.

5.4 Implementation

This section will discuss our implementation of row-based masked matvec, column-based matvec (masked and unmasked), and our direction-optimization heuristic. For simplicity in the following discussion, we use $\mathbf{m}(i)$ to denote checking whether the mask is nonzero, and if so, allowing the value to pass through to the output if it is. $\neg\mathbf{m}(i)$, while not discussed, does the inverse.

5.4.1 Row-based masked matvec (Pull phase)

Our parallel row-based masked matvec on the GPU is listed in Algorithm 8 and illustrated in Figure 5.3e. We parallelize over threads and have each thread check the $\mathbf{m}$. If $\mathbf{m}(i)$ passes

the check, the thread i checks its neighbours j in the matrix $\mathbf{A}^T(i,:)$ and tallies up the result if and only if the $\mathbf{v}(j)$ is also nonzero. For semirings with Boolean operators that support short-circuiting such as the one used for BFS, namely the Boolean semi-ring $(\{0,1\}, AND, OR, 0)$, once a single non-zero neighbour j is discovered (meaning it has been visited before), the thread can immediately write its result to the output vector and exit.

Algorithm 8 Masked row-based matrix multiplication over generalized semiring $(\mathbb{D}, \otimes, \oplus, \mathbb{I})$. The Boolean variable scmp controls whether or not $\mathbf{m}$ or $\neg\mathbf{m}$ is used.

1: **procedure** ROW_MASKED_MXV(Vector $\mathbf{v}$, Graph $\mathbf{A}^T$, MaskVector $\mathbf{m}$, MaskIdentity identity, Boolean scmp)
2:     **for** each thread i **in parallel do**
3:         **if** $\mathbf{m}(i) \neq$ identity XOR scmp **then**
4:             value $\leftarrow \mathbb{I}$
5:             **for** index j in $\mathbf{A}^T(i,:)$ **do**
6:                 **if** $\mathbf{v}(j) \neq 0$ **then**
7:                     value $\leftarrow$ value $\oplus \mathbf{A}^T(i,j) \otimes \mathbf{v}(j)$
8:                     **break** (optional: *early-exit* opt. enables this break)
9:                 **end if**
10:            **end for**
11:            $\mathbf{w}(i) \leftarrow$ value
12:        **end if**
13:    **end for**
14: **return** $\mathbf{w}$
15: **end procedure**

5.4.2 Column-based masked matvec (Push phase)

Our column-based masked matvec follows Gustavson's algorithm for SpGEMM (sparse matrix-sparse matrix multiplication), but specialized to matvec [37]. The key challenge in parallelizing Gustavson's algorithm is solving the multiway merge problem [3]. For the GPU, our parallelization approach follows the scan-gather-sort approach outlined by Yang et al. [70] and is shown in Algorithm 9. Instead of doing the multiway merge by doing $\mathcal{O}(nnz(m_{\mathbf{f}}^{+}) \log nnz(\mathbf{f}))$, we concatenate all lists and use radix sort, because radix sort tends to have better performance on GPUs. Our complexity then becomes $\mathcal{O}(nnz(m_{\mathbf{f}}^{+}) \log M)$, where M is the number of rows in the matrix; an increase in M forces us to do a higher bit radix sort.

We begin by computing the requisite space to leave in the output frontier for each neighbour list expansion. In compressed sparse row (CSR) format, node i computes its required space by taking the difference between the i-th and i+1-th row pointer values. Once each thread has its requisite length, we perform a prefix-sum over these lengths. This is fed into a higher-level abstraction, INTERVALGATHER, from the ModernGPU library [8]. On the prefix-sum array, INTERVALGATHER does a parallel search on sorted input to determine the indices from which each thread must gather. This gives us a load-balanced way of reading the column indices and values (Lines 6–9 in Algorithm 9).

During this process, the vector value of the corresponding thread $\mathbf{v}(i)$ is also gathered from global memory. This will allow us to multiply all of i's neighbours with $\mathbf{v}(i)$ using the $\otimes$ op-

erator. Once this is done, we write the column indices and multiplied values to global memory. Then we run a $\log M$-bit radix sort, where M is the number of matrix rows. One advantage of the structure-only optimization is that it allows us to cut down on the runtime, because this radix sort is often the bottleneck of the column-based masked matvec.

After the radix sort, a segmented reduction using the operator $(\oplus, \mathbb{I})$ gives us the temporary vector. The unmasked column-based matvec ends here. The masked version additionally filters out the values not in the mask by checking $\mathbf{m}(i)$.

Algorithm 9 Masked column-based matrix multiplication over generalized semiring $(\mathbb{D}, \otimes, \oplus, \mathbb{I})$. The Boolean variable scmp controls whether or not $\mathbf{m}$ or $\neg\mathbf{m}$ is used.

1: **procedure** COL_MASKED_MXV(Vector $\mathbf{v}$, Graph $\mathbf{A}^T$, MaskVector $\mathbf{m}$, MaskIdentity identity, Boolean scmp)
2: **for** each thread i **in parallel do**
3: length[i] ← row_ptr(i+1)-row_ptr(i+1) for all i such that $\mathbf{v}(i) \neq \mathbb{I}$
4: **end for**
5: scan ← prefix-sum length
6: addr[i] ← INTERVALGATHER(scan, $\mathbf{v}$)
7: col ← col_ind j such that $\mathbf{A}^T(j, i) \neq \mathbb{I}$ from addr[i]
8: val ← $\mathbf{A}^T(j, i)$ from addr[i]
9: val ← val $\otimes \mathbf{v}(i)$
10: write (col, val) to global memory
11: key-value sort (col, val)
12: (optional: *structure only* opt. turns this into a key-only sort)
13: segmented-reduction using $(\oplus, \mathbb{I})$ produces $\mathbf{w}'$
14: **for** each thread i **in parallel do**
15: ind ← ind such that $\mathbf{w}'(ind) \neq \mathbb{I}$
16: **if** $\mathbf{m}(ind) \neq$ identity XOR scmp **then**
17: $\mathbf{w}(i) = \mathbf{w}'(i)$
18: **else**
19: $\mathbf{w}(i) = \mathbb{I}$
20: **end if**
21: **end for**
22: **return** $\mathbf{w}$
23: **end procedure**

5.4.3 Direction-optimization heuristic

Implementing an efficient DOBFS requires good decisions to switch between forward and reverse. Beamer et al. compute the number of edges a column-based-with-mask implementation would have to touch as $nnz(m_\mathbf{f})$, and compare it with the number of edges a row-based-with-mask implementation would have to touch $nnz(m_\mathbf{u})$. If this ratio exceeds a constant determined by the heuristic and the ratio has increased from before, then the implementation switches from push to pull.

Beamer et al. proposed a heuristic to switch from push to pull when $\frac{nnz(m_\mathbf{f})}{nnz(m_\mathbf{u})} < \alpha$ for some factor α, and to switch back when $\frac{nnz(\mathbf{f})}{M} < \beta$ for some factor β [10]. We aim to match their

intent but face two difficulties: (1) GraphBLAS matvec calls do not pass in iteration counter information, and (2) we wish to avoid computing m_f speculatively. Instead, our method relies on the fact that $nnz(m_{\mathbf{f}}) \approx d\, nnz(\mathbf{f})$, where d is the average row length of the matrix and M is the number of rows in the matrix. If we also assume that $nnz(m_{\mathbf{u}}) \approx nnz(\mathbf{A}) \approx dM$ when we desire to switch, we see that $r = \frac{nnz(m_{\mathbf{f}})}{nnz(m_{\mathbf{u}})} \approx \frac{d\, nnz(\mathbf{f})}{dM} = \frac{nnz(\mathbf{f})}{M}$. Our method thus reduces to $\alpha = \beta$; if it is increasing and $\alpha = \beta > r$, we switch from push to pull, and if it is decreasing and $\alpha = \beta < r$, then we switch from pull to push. In this paper, we use $\alpha = \beta = 0.01$, which is optimal (defined by comparing with minimum of per-iteration push-only and pull-only BFS) for graphs we studied except 'i04' and the 3 non-scale free graphs, whose optimal BFS is push-only for all iterations.

To decide which version of matvec to use, we call the CONVERT function on the input vector $\mathbf{f}$. Then the vector tests whether $\mathbf{f}$ is stored in DenseVector or SparseVector format. If the former, then it checks whether the number of nonzeroes is low enough to warrant converting to a SparseVector using DENSE2SPARSE and whether it has decreased from the last time CONVERT was called on it. If the latter, it checks whether the number of nonzeroes is high enough to warrant converting to a DenseVector using SPARSE2DENSE and whether it has increased since the last time CONVERT was called. The user can select this sparse/dense switching point by passing in a floating-point value through the Descriptor of the matvec call. The default switchpoint is when the ratio of nonzeroes in the sparse matrix exceeds 0.01. Another way of expressing this is that once we have visited 1% of vertices in the graph in a BFS, we are sure to have hit a supernode.

As mentioned in Section 5.1, it is more efficient to store the input vector as a DenseVector object for row-based matvec. Similarly, it is efficient to store the frontier vector as a SparseVector. Therefore switching from push-to-pull in our implementation means converting the input vector from sparse to dense, and vice versa for pull-to-push. Using these SparseVector and DenseVector objects, we have the function signatures of following operations, which correspond to the four variants analyzed in Table 5.1:

1. ROW_MXV(DenseVector w, GrB_NULL, Matrix A, DenseVector v)

2. ROW_MASKED_MXV(DenseVector w, DenseVector mask, Matrix A, DenseVector v)

3. COL_MXV(SparseVector w, GrB_NULL, Matrix A, SparseVector v)

4. COL_MASKED_MXV(SparseVector w, DenseVector mask, Matrix A, SparseVector v)

There are already many efficient implementations of Operation (1) on the GPU [8, 12, 51], so we call ModernGPU's SpmvCsrBinary, but we implemented the other 3 operations ourselves.

The UML (Unified Modeling Language) diagram for the relationship between the backend implementation and the public interface is shown in Figure 5.6. The separation between the frontend interface Vector and the backend interface backend::Vector is inspired by the work of Zhang et al. in their GraphBLAS Template Library (GBTL) work [73]. Our work differs from theirs in two ways in that: (1) their implementation was intended as a proof-of-concept in programming language research rather than in performance; (2) our work demonstrates that

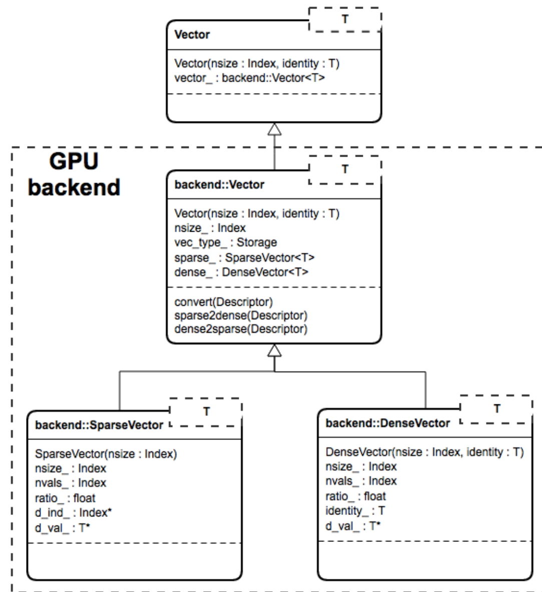


Figure 5.6: UML diagram showing the dynamic polymorphism we are using for Vector object. backend::SparseVector and DenseVector are two implementations of the backend::Vector GPU interface.

breaking up the Vector object into subclasses is necessary in order to take advantage of direction optimization; and (3) we program directly in CUDA while they program using Thrust, so they are limited by not being able to choose when to perform memory allocation. Our implementation enjoys a 31.8x geomean speed-up over GBTL over 6 datasets listed in their paper.

5.5 Experimental Results

5.5.1 Experimental setup

We ran all experiments in this paper on a Linux workstation with 2× 3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, 556 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVIDIA's nvcc compiler (version 8.0.61). The C code was compiled using gcc 4.9.4. Ligra was compiled using icpc 15.0.1 with CilkPlus. All results ignore transfer time (from disk-to-memory and CPU-to-GPU). The gather, scan, row-based mxv (without mask) operations are from the Modern GPU library [8]. The radix sort and reduce operations are from the CUDA UnBound library (CUB) [50]. All BFS tests were run 10 times with the average runtime and MTEPS used for results.

The datasets we used are listed in Table 5.3. 'soc-orkut' (soc-ork), 'soc-LiveJournal1' (soc-lj), and 'hollywood-09' (h09) are three social-network graphs; 'indochina-04' (i04) is a crawled hyperlink graph from indochina web domains; and 'kron_g500-logn21' (kron), 'rmat_s22_e64' (rmat-22), 'rmat_s23_e32' (rmat-23), and 'rmat_s24_e16' (rmat-24) are three generated R-MAT (Recursive-MATrix) graphs. These eight datasets are scale-free graphs with diameters of less than 30 and unevenly distributed node degrees (80% of nodes have degree less than 64). The 'rgg_n_24' (rgg), 'roadNet_CA' (roadnet), and 'road_USA' (road_usa) datasets have large diameters with small and evenly distributed node degrees (most nodes have degree less than 12).

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
kron_g500-logn21	2.1M	182.1M	213,904	6	gs
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_CA	2M	5.5M	12	849	rm
road_USA	23.9M	577.1M	9	6809	rm

Table 5.3: Dataset Description Table. Graph types are: r: real-world, g: generated, s: scale-free, and m: mesh-like.

soc-ork is from Network Repository [60]; soc-lj, h09, i04, kron, roadNet_CA, and road_usa are from the UF Sparse Matrix Collection [26]; and rmat-22, rmat-23, rmat-24, and rgg are randomized graphs we generated. All datasets have been converted to undirected graphs. Self-loops and duplicated edges are removed.

5.5.2 Graph framework comparison

As a baseline for comparison, we use the push-based BFS on the GPU by Yang et al. [70], because it is based in linear algebra and is (relatively) free of graph-specific optimizations. It does not support DOBFS. We also compare against four other graph frameworks (1 linear-algebra-based, 3 native-graph). SuiteSparse is a single-threaded CPU implementation of GraphBLAS. It is notable for being the first GraphBLAS implementation that adheres closely to the specification [25]. SuiteSparse performs matvecs by doing the column-based algorithm. CuSha is a vertex-centric framework on the GPU using the gather-apply-scatter (GAS) programming model [43]. Ligra is a CPU-based vertex-centric framework for shared memory [61]. It is the fastest graph framework we found on a multi-threaded CPU and was the first work that generalized push-pull to traversal algorithms other than BFS. Gunrock is a GPU-based frontier-centric framework [67] that generated the fastest single-processor BFS in our experiments.

5.5.3 Discussion of results

Figure 5.7 shows our performance results. In terms of runtime, we are 122×, 48.3×, 3.37×, 1.16× faster in the geomean than SuiteSparse, CuSha, the baseline, and Ligra respectively. We are 34.6% slower than Gunrock. Our implementation is relatively better on scale-free graphs, where we are 3.51× faster than Ligra on the scale-free datasets. In comparison, we are 3.2× slower than Ligra on the road maps and mesh graph. Our performance with respect to Gunrock is similar in that we do poorly on road maps (3.15× slower) compared with scale-free graphs (1.09× slower). This supports our intuition in Section 5.2 that DOBFS is helpful mainly on scale-free graphs.

The four biggest differences between Gunrock's and our implementation is that on top of

Dataset	Runtime (ms) [lower is better]						Edge throughput (MTEPS) [higher is better]					
	SuiteSparse	CuSha	Baseline	Ligra	Gunrock	This Work	SuiteSparse	CuSha	Baseline	Ligra	Gunrock	This Work
soc-ork	2165	244.9	122.4	26.1	**5.573**	7.280	98.24	868.3	1722	8149	**38165**	29217
soc-lj	1483	263.6	51.32	42.4	**14.05**	14.16	57.76	519.5	1669	2021	**6097**	6049
h09	596.7	855.2	23.39	12.8	**5.835**	7.138	188.7	131.8	4814	8798	**19299**	15775
i04	1866	17609	**71.81**	157	77.21	80.37	159.8	22.45	**4151**	1899	3861	3709
kron	1694	237.9	108.7	18.5	4.546	**4.088**	107.5	765.5	1675	9844	40061	**44550**
rmat-22	4226	1354	OOM	22.6	**3.943**	4.781	114.3	369.1	OOM	21374	**122516**	101038
rmat-23	6033	1423	OOM	45.6	**7.997**	8.655	83.81	362.7	OOM	11089	**63227**	58417
rmat-24	8193	1234	OOM	89.6	16.74	**16.59**	63.42	426.4	OOM	5800	31042	**31327**
rgg	230602	68202	9147	918	**593.9**	2991	1.201	3.887	30.28	288.8	**466.4**	92.59
roadnet	342	288.5	284.9	**82.1**	130.9	214.4	16.14	14.99	19.38	**67.25**	42.18	25.75
road_usa	9413	36194	26594	978	**676.2**	7155	6.131	7.944	2.17	59.01	**85.34**	8.065

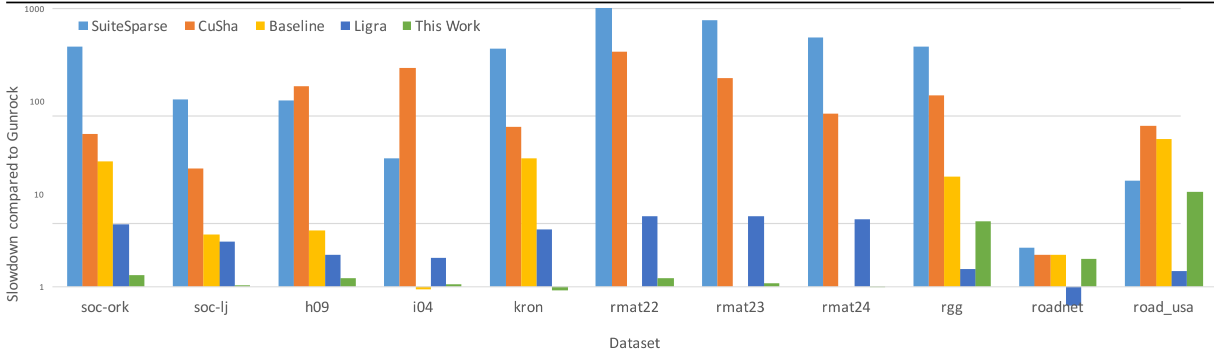


Figure 5.7: The table compares our work to other graph libraries (SuiteSparse [25], CuSha [43], a baseline push-based BFS [70], Ligra [61], and Gunrock [67]) implemented on 1× Intel Xeon CPU and 1× Tesla K40c GPU. Bold is fastest for that dataset. OOM means out of memory. The graph shows the same data presented as a slowdown compared to Gunrock.

the optimizations discussed in this paper, they also employ (1) local culling, (2) keeping sparse vector indices in unsorted order with possible duplicates, (3) kernel fusion, and (4) a different traversal strategy for road networks.

Local culling Instead of our write to global memory in Line 9 of Algorithm 9 which is followed by an expensive key-value sort, Gunrock's filter step incorporates a series of inexpensive heuristics [52] 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.

Unsorted order and redundant elements When performing the column-based masked matvec as in Figure 5.3d, our complexity is $\mathcal{O}(nnz(m_{\mathbf{f}}^{+}) \log M)$, so the bottleneck is in making the elements unique. If duplicate indices are tolerated, we can omit the multiway merge entirely, and get rid of the logarithmic factor leaving us with $\mathcal{O}(nnz(m_{\mathbf{f}}^{+}))$. While redundant vertices impose an extra cost, BFS is an algorithm that can tolerate redundant vertices and in some cases, it may be cheaper to allow a few extra vertices to percolate through the computation than to go to significant effort to filter them out. This approach may not be suitable for GraphBLAS, because such an optimization may be too BFS-focused and would generalize poorly.

Kernel fusion Because launching a GPU kernel is relatively expensive, optimized GPU programs attempt to fuse multiple kernels into one to improve performance ("kernel fusion"). Gunrock fuses kernels in several places, for example, fusing Lines 7 and 8 in Algorithm 7 during pull traversal. This optimization may be a good fit for a non-blocking implementation of GraphBLAS, which would construct a task graph and fuse tasks when it deemed worth fusing to improve performance.

Different traversal strategy for road networks For road networks, Gunrock uses the TWC (Thread Warp CTA) load-balancing mechanism of Merrill et al. [52]. TWC is cheaper to apply than other load-balancing mechanisms, which makes it a good match for road networks that have many BFS iterations each with little work.

5.6 Conclusion

In this chapter we demonstrated that push-pull corresponds to the concept of column- and row-based masked matvec. We analyzed four variants of matvec, and show theoretically and empirically they have fundamentally different computational complexities. We presented evidence that there is a difference in complexity between doing a matvec on arbitrary input compared to doing matvec on a BFS frontier. We provided experimental evidence that the concept of a mask to take advantage of *output sparsity* is critical for a linear-algebra based graph analytic framework to be competitive with state-of-the art vertex-centric graph frameworks on the GPU and multi-threaded CPU.

In the next chapter, we try to use our knowledge about SpMV in order to improve performance of mulitple column vectors (sparse matrix-dense matrix multiplication).

Chapter 6

Design Principles for Sparse Matrix Multiplication on the GPU

In this chapter, we take a turn to improve the state-of-the-art in sparse matrix-dense matrix (SpMM) multiplication.

Our main contributions in this work are:

1. We generalize two main classes of SpMV algorithms—(1) row splitting and (2) merge-based—for the SpMM problem and implement them on the GPU.

2. We introduce a simple heuristic that selects between the two kernels with an accuracy of 95.9% compared to optimal.

3. Using our multi-algorithm and heuristic, we achieve a geomean speed-up of 23.5% and up to a maximum of 3.6x speed-up over state-of-the-art SpMM implementations over 195 datasets from the SuiteSparse Matrix Collection [26].

6.1 Design Principles

We discuss two design principles that every irregular problem on the GPU must follow for good performance. Ideally, we attain *full utilization* of the GPU hardware, where a ready warp can be run on every cycle, all computational units are doing useful work on every cycle, and all memory accesses are coalesced. Our principles for reaching this goal are (1) effective latency-hiding through a combination of thread- and instruction-level parallelism (TLP and ILP) and (2) efficient load-balancing. Then we will look at state-of-the-art SpMM implementations to understand their inefficiencies.

1. Load-balancing

2. Latency-hiding ability

    (a) Thread-level parallelism (TLP)

    (b) Instruction-level parallelism (ILP)

[2]This chapter substantially appeared as "Design Principles for Sparse Matrix Multiplication on the GPU" [71], for which I was responsible for most of the research and writing.

6.1.1 Latency hiding with TLP and ILP

Memory operations to a GPU's main memory take hundreds of clock cycles. The GPU's chief method for hiding the cost of these long-latency operations is through thread-level parallelism (TLP). Effective use of TLP requires that the programmer give the GPU enough work so that when a GPU warp of threads issues a memory request, the GPU scheduler puts that warp to sleep and another ready warp becomes active. If enough warps are resident on the GPU (if we have enough TLP), switching between warps can completely hide the cost of a long-latency operation. We quantify the amount of TLP in a program as *occupancy*, the ratio of available (issued) warps to the maximum number of warps that can be supported by the GPU. Higher occupancy yields better latency-hiding ability, which allows us to approach full utilization.

Another latency-hiding strategy is exploiting instruction-level parallelism (ILP) and its ability to take advantage of overlapping the latency of multiple memory operations within a single thread. Because the GPU's memory system is deeply pipelined, a thread can potentially issue multiple independent long-latency operations before becoming inactive, and those multiple operations will collectively incur roughly the same latency as a single operation. While this yields a significant performance advantage, it relies on the programmer exposing independent memory operations to the hardware. We can achieve this goal by assigning multiple independent tasks to the same thread ("thread coarsening").

GPUs have a fixed number of registers. TLP requires many resident warps, each of which requires registers. ILP increases the work per thread, so each thread requires more registers. Thus TLP and ILP are in opposition, and attaining full utilization requires carefully balancing both techniques. While TLP is commonly used across all of GPU computing, ILP is a less explored area, with prior work limited to dense linear algebra [64] and microcode optimization [38].

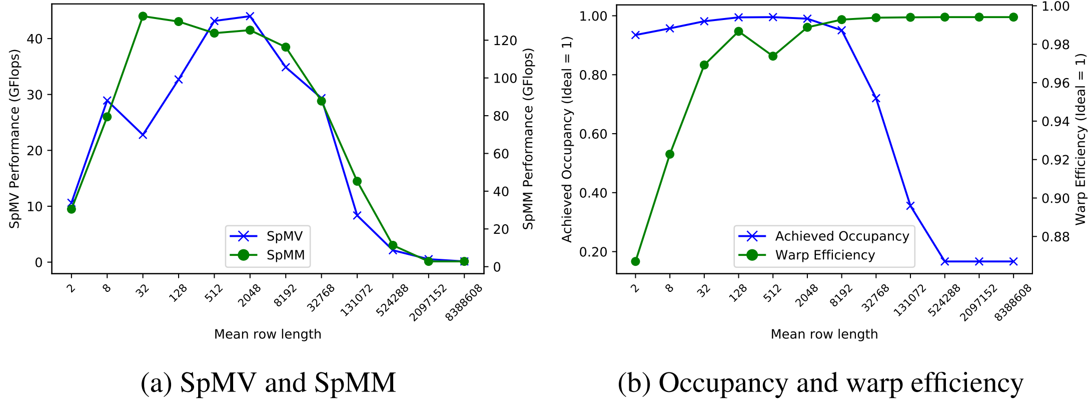
6.1.2 Load-balancing

We now turn to the problem of ensuring that all computational units are doing useful work on every cycle, and that the memory accesses from those warps are coalesced to ensure peak memory performance. In the context of SpMV and SpMM, this "load-balancing" problem has two aspects:

1. Load imbalance *across* warps. Some CTAs or warps may be assigned less work than others, which may lead to these less-loaded computation units being idle while the more loaded ones continue to do useful work. In this paper, we term this "type 1" load imbalance.

2. Load imbalance *within* a warp, in two ways, which we collectively call "type 2" load imbalance. (a) Some warps may not have enough work to occupy all 32 threads in the warp. In this case, thread processors are idle, and we lose performance. (b) Some warps may assign different tasks to different threads. In this case, SIMD execution within a thread means that some threads are idle while other threads are running; moreover, the divergence in execution across the warp means memory accesses across the entire warp are unlikely to be coalesced.

For irregular matrices, we claim that SpMV and SpMM are fundamentally load-balancing problems on the GPU. As evidence, Figure 6.1 shows load imbalance in a vendor-supplied im-



(a) SpMV and SpMM (b) Occupancy and warp efficiency

Figure 6.1: Synthetic benchmark showing NVIDIA cuSPARSE SpMV and SpMM performance as a function of matrix dimensions on a Tesla K40c, and SpMM's achieved occupancy and warp efficiency (inverse of divergence).

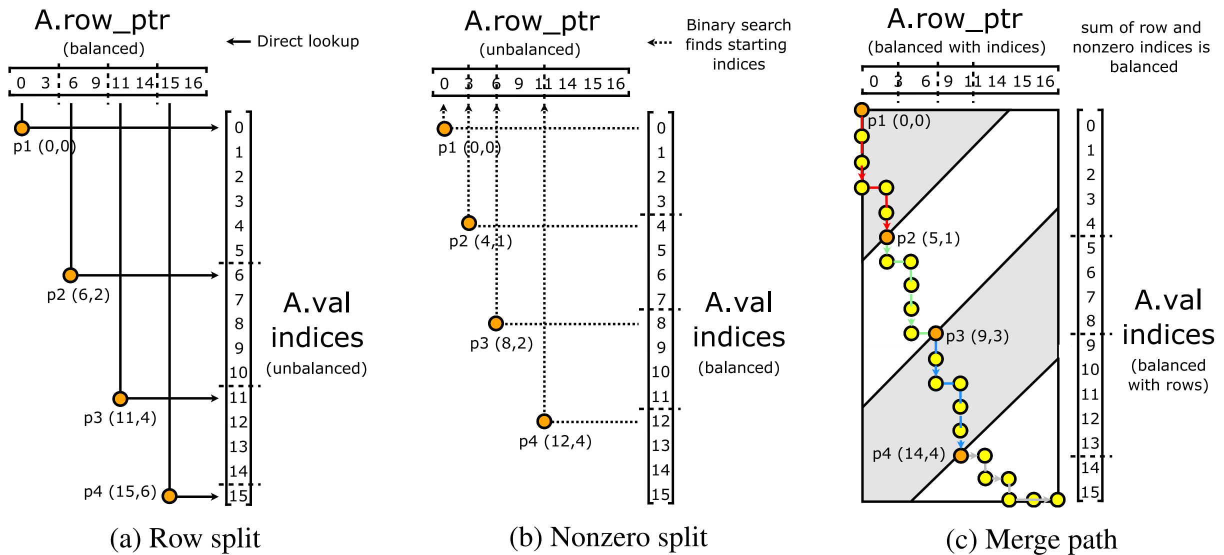
plementation from a synthetic benchmark. The experimental setup is described in Section 5.5. The right side of the x-axis represents Type 1 load imbalance, where long matrix rows are not divided enough, resulting in some computation resources on the GPU remaining idle while others are overburdened. The left size of the x-axis represents Type 2 load imbalance where too many computational resources are allocated to each row, so some remain idle.

6.2 Parallelizations of CSR SpMM

This section reviews three existing parallelizations of SpMV through the lens of the design principles from Section 6.1. While our implementations of SpMM share some characteristics with SpMV parallelizations, we also faced several different design decisions for SpMM, which we discuss below. The three SpMV variants are illustrated in Figure 6.2 and summarized here:

1. Row split [12]: 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 [8, 23]: Assign an equal number of nonzeroes per processor. Then do a 1-D (1-dimensional) binary search on *row offsets* to determine at which row to start.
   (b) Merge path [51]: Assign an equal number of {nonzeroes and rows} per processor. This is done by doing a 2-D binary search (i.e., on the diagonal line in Figure 6.2c) over *row offsets* and *nonzero indices* of matrix $\mathbf{A}$.

While row split focuses primarily on ILP and TLP, nonzero split and merge path focus on load-balancing as well. We consider nonzero split and merge path to be *explicit load-balancing* methods, because they rearrange the distribution of work such that each thread must perform T independent instructions; if $T > 1$, then explicit load-balancing creates ILP where there was previously little or none. Thus load-balance is closely linked with ILP, because if each thread is



(a) Row split (b) Nonzero split (c) Merge path

Figure 6.2: The three parallelizations for CSR SpMV and SpMM on matrix $\mathbf{A}$. The orange markers indicate segment start for each processor ($P = 4$).

guaranteed $T > 1$ units of independent work (ILP), then each thread is doing the same amount of work (i.e., is load-balanced).

6.2.1 Algorithm I: Row-splitting SpMM

Row split aims to assign each row to a different thread, warp, or CTA. Figure 6.3a shows the warp assignment version. The typical SpMV row split is only the left-most column of matrix $\mathbf{B}$ with orange cells replaced by green cells. This gives SpMV 1 independent instruction and uncoalesced, random accesses into the vector. Although row-split is a well-known method for SpMV [12], we encountered three important design decisions when extending it to SpMM:

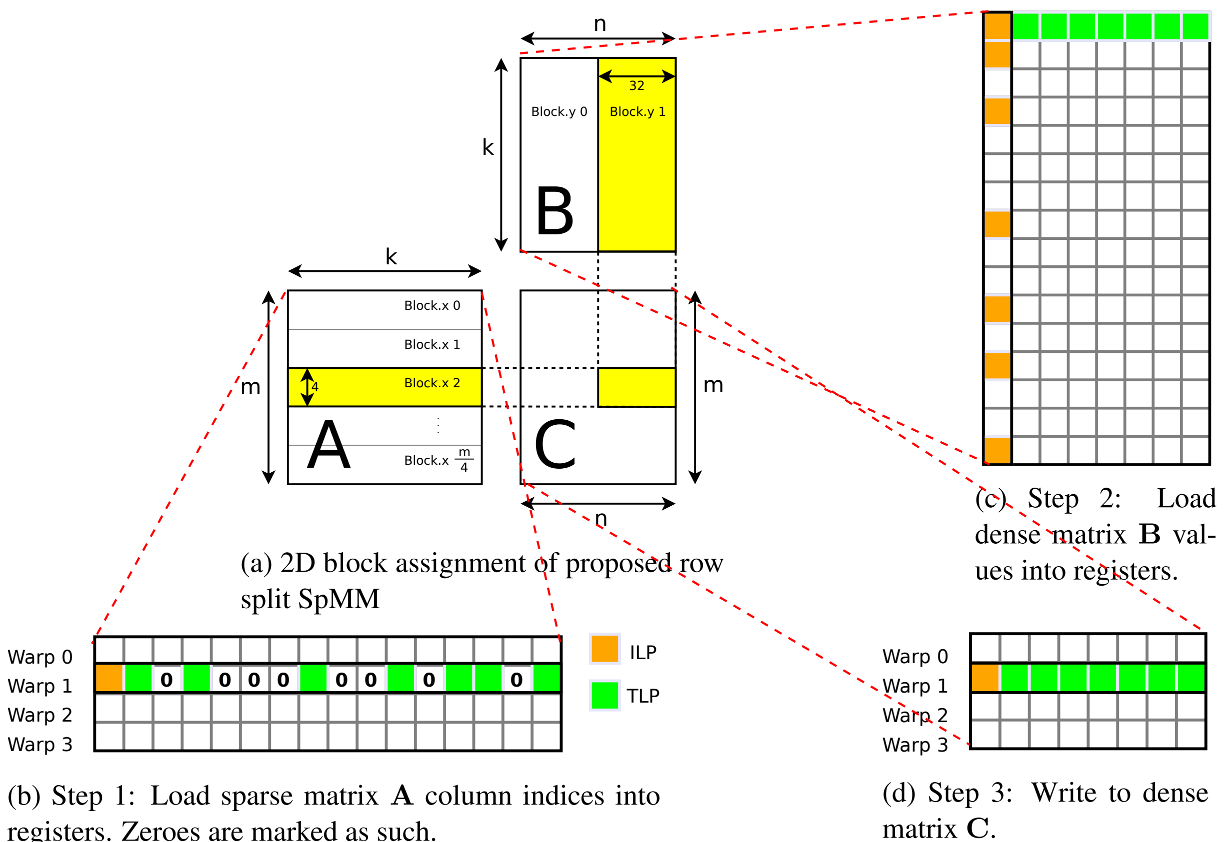
1. Granularity. We assigned each row to a warp compared to the alternatives of assigning a thread and a CTA per row. This leads to the simplest design out of the three options, since it gives us coalesced memory accesses into $\mathbf{B}$. For matrices with few nonzeroes per row, the thread-per-matrix-row work assignment may be more efficient. This is borne out by Figure 6.4.

2. Memory access pattern. This design decision had the greatest impact on performance. To our knowledge, this is the first time in literature this novel memory access strategy has been described. Our thread layout is shown in Figure 6.3c. For SpMM, we have two approaches we could take: each thread is responsible for loading a column or a row of the matrix $\mathbf{B}$.

We discovered the first approach is better, because the memory accesses into $\mathbf{B}$ are independent and can be done in a coalesced manner (provided that $\mathbf{B}$ is in row-major order). In contrast, memory accesses into a column-major $\mathbf{B}$ would be independent but uncoalesced. Compared to the SpMV case, each thread now has 32 independent instruction and coalesced memory accesses into $\mathbf{B}$, which significantly amortizes the cost of memory accesses compared to accessing a single vector. However, since we are forcing threads to pass a dummy column index if they are out of bounds within a row, the effective number of independent instruction and coalesced memory accesses is sensitive to row lengths that do not divide 32. For example,

if the row length is 33, then we will be doing 64 independent instruction and coalesced memory accesses into $\mathbf{B}$. Whether or not they divide 32 does not matter for very long rows, because the cost is amortized by efficiently processing batches of 32. However, we would expect row split to be negatively impacted by Type 2 load imbalances. The summary of this ILP analysis is shown in Table 6.1.

3. Shared memory. The key component required is a round of 32 broadcasts (using the "shuffle" warp intrinsic __shfl) by each thread to inform all other threads in the warp which $\mathbf{B}$ row ought to be collectively loaded by the entire warp. This is required or otherwise each thread would be responsible for loading its own row, which would result in uncoalesced access. We could have also implemented this using shared memory, but since all threads are within a single warp, there is no disadvantage to preferring warp intrinsics. That they are within a single warp is a consequence of our decision to assign each row to a warp rather than a CTA.



(a) 2D block assignment of proposed row split SpMM

(b) Step 1: Load sparse matrix $\mathbf{A}$ column indices into registers. Zeroes are marked as such.

(c) Step 2: Load dense matrix $\mathbf{B}$ values into registers.

(d) Step 3: Write to dense matrix $\mathbf{C}$.

Figure 6.3: Figure 6.3a shows the tiling scheme we use. Figures (b), (c), (d) represent the yellow blocks from Figure 6.3a. Row split SpMM ILP (orange) and TLP (green) are shown using warp 1 with 8 threads per warp. In practice, we use 32 threads and 4 warps per GPU cooperative thread array (CTA). Matrix $\mathbf{A}$ is sparse in CSR format. Matrices $\mathbf{B}$ and $\mathbf{C}$ are both dense in row-major format.

Table 6.1: This table shows the number of independent instructions per GPU thread for SpMV and SpMM with default value shown in brackets, as well as the register usage and the extra number of memory accesses with respect to the row-split algorithm. T is the number of work items per thread (typically specified as a tuning parameter to the algorithm). L is the number of nonzeroes modulus 32 in the row of $\mathbf{A}$ that we are computing. B is the CTA size. Typical values for T in SpMV and SpMM are 7 and 1 respectively, while a typical value for B is 128. T cannot be set arbitrarily high, because high register usage causes lower occupancy. $\mathbf{A}.nnz$ is the number of nonzeroes in the sparse matrix $\mathbf{A}$. $\mathbf{B}.ncols$ is the number of columns of the dense matrix $\mathbf{B}$.

	SpMV		SpMM	
Operation	Row-split	Merge-based	Row-split	Merge-based
Read $\mathbf{A}$.col_ind and $\mathbf{A}$.val	1	$T(7)$	1	$T(1)$
Read $\mathbf{x}$ / Read $\mathbf{B}$	1	$T(7)$	$0 < L \leq 32$	$32T(32)$
Write $\mathbf{y}$ / Write $\mathbf{C}$	1	$T(7)$	1	$32T(32)$
Register usage	2	$2T(14)$	64	$64T(64)$
Memory access overhead	0	$\frac{\mathbf{A}.nnz}{B \times T}$ $(\frac{\mathbf{A}.nnz}{896})$	0	$\frac{\mathbf{B}.ncols \times \mathbf{A}.nnz}{B \times T}$ $(2\mathbf{A}.nnz)$

6.2.2 Algorithm II: Merge-based SpMM

The essence of merge-based algorithms is to explicitly and evenly distribute the nonzeroes across parallel processors. It does so by doing a two-phase decomposition: In the first phase (PARTITIONSPMM), it divides T work amongst all threads, and based on this deduces the starting indices of each CTA. Once coordinated thusly, work is done in the second phase. In theory, this approach should eliminate both Type 1 and Type 2 load imbalances, and performs well in recent SpMV implementations [51]. We made the following design decisions when generalizing this technique to SpMM:

1. *Memory access pattern.* For fetching $\mathbf{B}$, we adapt the memory access pattern that was successful in row-splitting. However, here, we must first apply the first phase (i.e., PARTITION-SPMM, Line 2 of Algorithm 10) to tell us the rows each CTA ought to look at if we want an equal number of nonzeroes per CTA. Then, we can apply the broadcast technique to retrieve $\mathbf{B}$ values using coalesced accesses.

2. *Register usage.* Since we opted for the coalesced memory access pattern explained in the row-splitting section, we require 32$\times$ the number of registers in order to store the values. Due to this limitation, the number of independent instructions per thread T is limited to 1, so we see no further latency-hiding gain from ILP over that of row-split.

3. *Memory access overhead.* There are two sources of memory access overhead compared to the row-splitting algorithm: (1) the additional GPU kernel that determines the starting rows for each block (Line 3), and (2) the write of the carry-out to global memory for matrix rows of $\mathbf{C}$ that cross CTA boundaries (Lines 29 and 33). Since the user is unable to synchronize CTAs in CUDA, this is the only way the user can pass information from one CTA to another. The first source of additional memory accesses is less of a problem for SpMM compared to SpMV,

Algorithm 10 The merge-based SpMM algorithm.

Input: Sparse matrix in CSR $\mathbf{A} \in \mathbb{R}^{m \times k}$ and dense matrix $\mathbf{B} \in \mathbb{R}^{k \times n}$.
Output: $\mathbf{C} \in \mathbb{R}^{m \times n}$ such that $\mathbf{C} \leftarrow \mathbf{AB}$.

```
 1: procedure SpmmMerge(A, B)
 2:     limits[] ← PartitionSpmm(A, blockDim.x)          ▷ Phase 1: Divide work and run binary-search
 3:     for each CTA i in parallel do                      ▷ Phase 2: Do computation
 4:         num_rows ← limits[i + 1] − limits[i]
 5:         shared.csr ← GlobalToShared(A.row_ptr + limits[i], num_rows)   ▷ Read A and store to shared memory
 6:         end ← min(blockDim.x, A.nnz - blockIdx.x × blockDim.x)
 7:         if row_ind < end then
 8:             col_ind ← A.col_ind[row_ind]               ▷ Read A if matrix not finished
 9:             valA ← A.values[row_ind]
10:         else
11:             col_ind ← 0                                ▷ Otherwise do nothing
12:             valA ← 0
13:         end if
14:         for each thread j in parallel do
15:             for j = 0, 1, . . . , 31 do                ▷ Unroll this loop
16:                 new_ind[j] ← Broadcast(col_ind, j)     ▷ Each thread broadcasts
17:                 new_val[j] ← Broadcast(valA, j)        ▷ col_ind and valA
18:                 valB[j] ← B[col_ind][j] × new_val[j]   ▷ Read B
19:             end for
20:         end for
21:         terms ← PrepareSpmm(shared.csr)                ▷ Flatten CSR-to-COO
22:         carryout[i] ← ReduceToGlobalSpmm(C, valB, valB)  ▷ Compute partial of C and save carry-outs
23:     end for
24:     FixCarryout(C, limits, carryout)   ▷ Carry-out fix-up (rows spanning across blocks)
25:     return C
26: end procedure
```

Table 6.2: A selection of 20 datasets from the 195 SuiteSparse matrices used in our experiments. d is the mean row length of the matrix. nnz is the number of nonzeroes in the matrix. The left datasets have mean row length with arithmetic mean 7.92, while the right have 62.5.
*: Despite having the same number of rows and nonzeroes, these two matrices differ in nonzero structure and are treated as independent matrices in our experiments.

Dataset	Description	d	nnz	Dataset	Description	d	nnz
wheel_601	combinatorial opt.	2.41	2.2M	StocF-1465	CFD problem	13.3	19.5M
neos3	linear programming	4.01	2.1M	ldoor	structural problem	47.9	45.6M
c-big	non-linear opt.	5.78	2M	bmw3_2	structural problem	48.7	11.1M
ins2	optimization problem	7.89	2.4M	boneS10	3D trabecular bone	59.6	54.6M
ASIC_320k	circuit simulation	8.19	2.6M	bmwcra_1	stiffness matrix	70.5	10.5M
web-Stanford*	web graph	8.2	2.3M	bone010	3D trabecular bone	71.6	70.7M
Stanford*	web graph	8.2	2.3M	inline_1	stiffness matrix	72.1	36.3M
citationCiteseer	citation network	8.62	2.3M	F1	stiffness matrix	77.1	26.5M
FullChip	circuit simulation	8.9	26.7M	audikw_1	structural problem	81.3	76.7M
circuit5M	circuit simulation	10.7	59.5M	crankseg_2	structural problem	220	14.1M

because they are amortized by the increased work. The second source, however, scales with the number of B columns. Thus we face a trade-off between having more efficient memory access pattern (assign 32 columns per CTA so memory access is coalesced), and having less memory access overhead (assign 4 columns per CTA so T can be set higher resulting in fewer CTA boundaries that need to be crossed). The first approach resulted in better performance.

6.3 Experimental Results

6.3.1 Experimental Setup

We ran all experiments in this paper on a Linux workstation with $2\times$ 3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, 256 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVIDIA's nvcc compiler (version 8.0.44). The C code was compiled using gcc 4.9.3. All results ignore transfer time (from disk-to-memory and CPU-to-GPU). The merge path operation is from the Modern GPU library [8]. The version of cuSPARSE used was 8.0.

The 195 datasets mentioned in the previous section represent a random sample from the SuiteSparse sparse matrix collection. The topology of the datasets varies from small-degree large-diameter (road network) to scale-free. In the microbenchmark Figure 6.1a, dense matrices (varying from 2 rows with 8.3M nonzeroes per row to 8.3M rows with 2 nonzeroes per row) used in the micro-benchmark are generated to be nonzero, and converted to CSR sparse matrix storage. We then multiply the matrix by a dense vector and a dense matrix with 64 columns using the vendor-supplied SpMV and SpMM implementations respectively.

6.3.2 Algorithm I: Row-split

Figure 6.5a shows the performance of our row split implementation on 10 SuiteSparse datasets with long matrix rows (62.5 nonzeroes per row on average). We obtain a geomean speed-up of 30.8% over the next fastest implementation and 39% peak improvement.

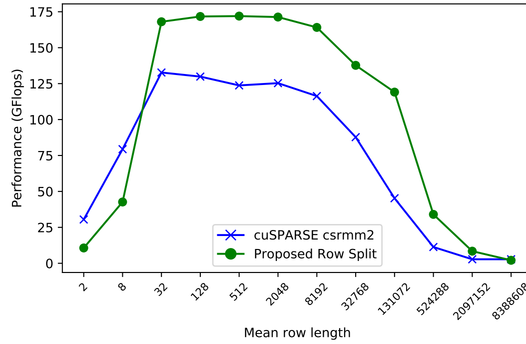
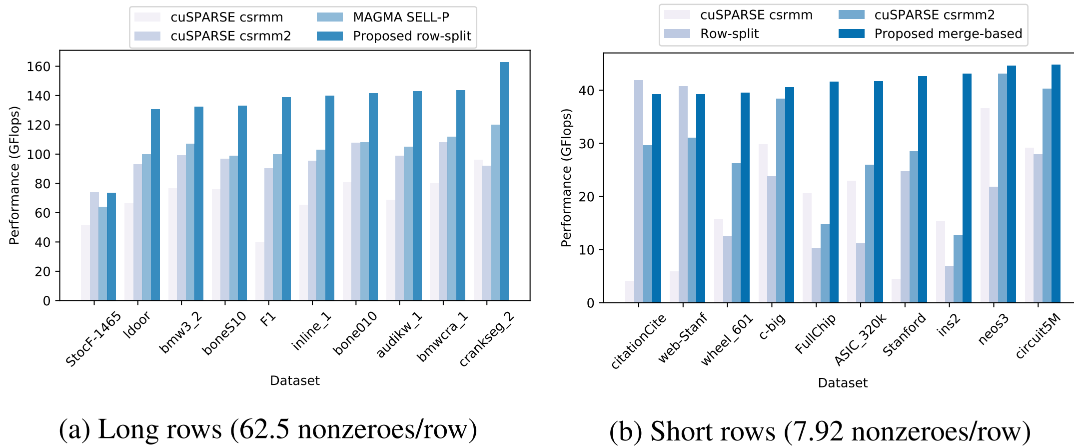


Figure 6.4: The performance of our proposed SpMM row split kernel vs. NVIDIA cuSPARSE's SpMM as a function of aspect ratio on a Tesla K40c.



(a) Long rows (62.5 nonzeroes/row)

(b) Short rows (7.92 nonzeroes/row)

Figure 6.5: Performance comparison between the proposed ILP-centric row split kernel and other state-of-the-art kernels on matrices with *long* and *short* row lengths on Tesla K40c using single-precision floating-point. cuSPARSE csrmm and csrmm2 are from a vendor-supplied library [54]. MAGMA SELL-P is by Anzt, Tomov, and Dongarra [2].

We suspect our performance drop to the left in Figure 6.4 comes from the sensitivity to parameter L on row lengths that are significantly less than 32. This causes divergence and uncoalesced memory accesses. On the right hand side, we do much better than cuSPARSE. We believe this is due to the additional occupancy that we can get from superior ILP, which is better at hiding latency. Using the profiler, we noted a 102% improvement in executed instructions per cycle for the matrix sized 128-by-131072.

6.3.3 Algorithm II: Merge-based

Figure 6.5b shows the performance of our merge-based SpMM kernel on 10 SuiteSparse datasets with short matrix rows (7.92 nonzeroes on average). We obtain a geomean speed-up of 53% over cuSPARSE csrmm2 and 237% peak improvement. We think the biggest reason that merge path is doing better than the other methods is because it handles Type 2 load imbalances much better. Other methods inevitably encounter warp efficiency degradation due to the divergence

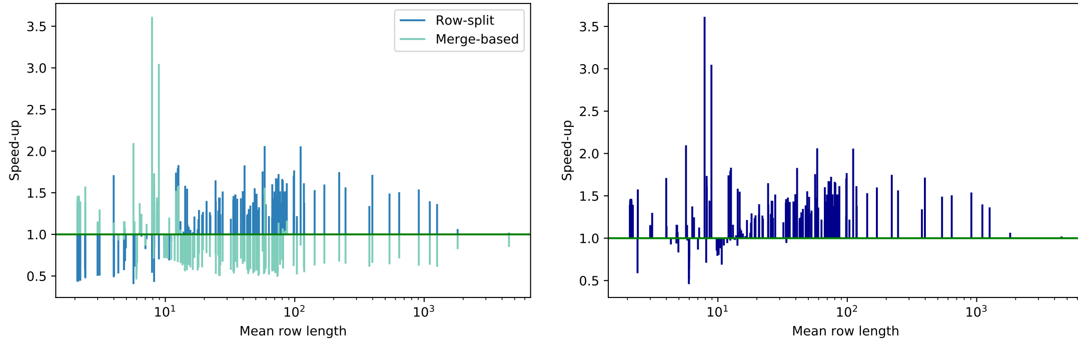
caused by short rows, as shown in Figure 6.1b. However, merge path can handle these short rows very well by simply allocating more rows to a CTA if the rows are short.

Another interesting observation to make is that the merge path throughputs in Figure 6.5b all tend to be lower than their row split equivalents. This means that merge path has more overhead than row split, so it is only worth it to perfectly load-balance matrices when it is profitable to do so. While Merrill and Garland found their merge-based solution was better than row split on SpMV [51], ours did not perform as well on SpMM, as explained in the next paragraph.

As Table 6.1 shows, merge path's advantage in SpMV comes from being able to obtain T times more ILP per thread than row split, but it enjoys no such advantage in SpMM,where row splitting gets as much ILP as there are nonzeroes in the sparse matrix row as long as row split can afford to pay the register cost. This can be seen in Figure 6.3a. While merge path has the opportunity to obtain T times more ILP, we discovered that we need to keep $T = 1$ in order to keep the register count manageable. In typical merge path SpMV implementations, T can be as high as 7. The ILP advantage merge-based had in SpMV is not so assured.

6.3.4 Heuristic

By comparing the speed-up of row split and merge-based to the fastest vendor-supplied SpMM on 195 SuiteSparse sparse matrix collection datasets [26] (see Figure 6.6a), we show that the two proposed algorithms achieve speed-ups over the SpMM state-of-the-art in separate regions on the spectrum of matrix irregularity. However, the geomean speed-up is only a 4.98% gain and 27.4% slowdown for row split and merge-based respectively.



(a) Row split and merge-based separately vs. cuSPARSE csrmm2.

(b) Combined row split and merge-based vs. cuSPARSE csrmm2.

Figure 6.6: Performance comparison between proposed row split kernel, proposed merge-based kernel, and cuSPARSE csrmm2 on 195 non-trivial datasets from the SuiteSparse sparse matrix collection [26].

Therefore, we propose a heuristic for switching between them using an inexpensive $O(1)$ calculation $d = \frac{nnz}{n}$. Our heuristic is simply computing the average row length for the matrix, and using this value to decide whether to use merge-based or row split. To pinpoint the transition point, we examine Figure 6.6a. For our heuristic, we decide that we will use merge-based on datasets whose mean row length is less than 9.35, and row split otherwise.

Using this heuristic, we obtain an overall 23.5% geomean speed-up, and up to a peak of

$3.6\times$, over the vendor-supplied library cuSPARSE csrmm2. Over cuSPARSE csrmm, we obtain a $2.52\times$ geomean speed-up and $22.4\times$ peak speed-up. The result is shown in Figure 6.6. Using this heuristic as a binary classifier, we get 95.9% accuracy vs. an oracle that perfectly chooses the fastest implementation.

6.4 Conclusion

In this paper we implement two promising algorithms for computing sparse matrix dense matrix multiplication on the GPU. Our results using SpMM show considerable performance improvement over the vendor-supplied SpMM on a wide spectrum of graphs. One of the keys to our high performance is our memory-access strategy that allows coalesced access into all 3 matrices (see Figure 6.3a).

In Figure 6.6, the points in the plot become sparser as the mean row length d gets larger. Greiner and Jacob have proven theoretically that as d exceeds some hardware threshold, namely $\frac{m}{M}$ where m is the number of rows in the sparse matrix and M is the size of the fast memory of the device, that the tiling algorithm used in nearly every SpMM implementation will stop being optimal [36]. Instead, they claim that tiling both the sparse matrix $\mathbf{A}$ and $\mathbf{B}$ in a manner akin to tiling dense matrix-matrix multiplication is optimal. In future work, it would be interesting to find empirical evidence of such a transition.

In the next chapter, we put all the pieces together and describe general design principles that if heeded, can be used to build a high-performance linear-algebraic graph framework on the GPU.

Chapter 7

Design of GraphBLAST

In this chapter, we are interested in answering the following question: What are the design principles required to build a GPU implementation based in 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 (graphics processing unit). Our implementation is for single GPU, but we believe the design we propose here will allow us to extend it to a distributed implementation with future work.

To perform a comprehensive evaluation of our system, we need to compare our framework against the state-of-the-art graph frameworks on the CPU and GPU, and hardwired GPU implementations, which are problem-specific GPU code that someone has handtuned for performance. The state-of-the-art graph frameworks we will be comparing against are Ligra [61] for CPU and Gunrock [67] for the GPU, which were described in detail in Chapter 3. The hardwired implementations will be Enterprise (BFS) [47], delta-stepping SSSP [24], pull-based PR [43], and bitmap-based triangle counting [14]. As described in Section 1, the graph algorithms we will be evaluating our system on are:

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

Our contributions in this paper are as follows:

1. We give a brief introduction of GraphBLAS's computation model (Section 7.1).
2. We demonstrate the importance of exploiting *input sparsity*, which means picking the algorithm that minimizes the number of computations and whose consequence is direction-optimization (Section 7.2).

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

3. We show the importance of exploiting *output sparsity*, which is implemented as masking and can be used to reduce the number of computations of several graph algorithms (Section 7.3).

4. We explain the load-balancing techniques required for high-performance on the GPU (Section 7.4).

5. We review how common graph algorithms are expressed in GraphBLAST (Section 7.5).

6. We show that enabled by the optimizations *exploiting sparsity*, *masking*, and *proper load-balancing* our system GraphBLAST gets $36\times$ geomean ($892\times$ peak) over SuiteSparse GraphBLAS for sequential CPU and $2.14\times$ geomean ($10.97\times$ peak) and $1.01\times$ ($5.24\times$ peak) speed-up over state-of-the-art graph frameworks on CPU and GPU respectively on several graph algorithms (Section 7.6).

7.1 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 spec [18] and selected papers [17, 42, 49] for a full treatment. At the end of this section, we give a running example (Section 7.1.9). In later sections, we will show how taking advantage of *input* and *output sparsity* will, even in the small running example, allow computation to complete in much fewer memory accesses.

GraphBLAS's model of computation consists Matrix, Vector, Operation, Semiring, Masking and Descriptor. The programmer first defines Vector and Matrix objects (Lines 12-13 of Algorithm 11 (right)), interacts with these objects in order to perform useful computation, and extracts the data from these objects. During the process of computation, the Vector and Matrix objects are assumed as being *opaque* to the user meaning no assumptions can be made regarding the data structures behind them.

7.1.1 Matrix

A Matrix is the adjacency matrix of a graph. A full list of methods used to interact with Matrix objects is shown in Table 7.2. When referring to matrices in mathematical notation, we will indicate it by uppercase boldface i.e. $\mathbf{A}$.

7.1.2 Vector

A Vector is the set of vertices in a graph that are currently acting as starting points in the graph search. 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 it by a lowercase boldface i.e. $\mathbf{x}$.

7.1.3 Operation

An Operation is a memory access pattern common to many graph algorithms. A full list of operations is shown in Table 7.2.

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`[1]	build dense vector or matrix	build vertex set or graph from tuples
`fill`[1]	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 7.1: A list of Matrix and Vector operations in GraphBLAST.
Note [1]: These are convenience operations not found in GraphBLAS specification, but were added by the authors for GraphBLAST.

7.1.4 Semiring

A semiring is the computation on vertex and edge 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. What the $(\oplus, \otimes, \mathbb{D}, \mathbb{I})$ represent are 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 tropical semiring) $(\oplus, \otimes, \mathbb{D}, \mathbb{I}) = \{\min, +, \mathbb{R} \cup \{+\infty\}, +\infty\}$, which can 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 $s \to u \to v$, potentially via many parent vertices u.

Operation	Math Equivalent	Description	Graph application
mxm	$\mathbf{C} = \mathbf{AB}$	matrix-matrix mult.	multi-source traversal
vxm	$\mathbf{w} = \mathbf{Au}$	matrix-vector mult.	single-source traversal
mxv	$\mathbf{w} = \mathbf{vA}$	vector-matrix mult.	single-source 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 7.2: A list of operations in GraphBLAST.

- $\oplus$: In MinPlus, $\oplus = \min$. What this operation means is choosing the distance from $s \to u \to v$ 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 i.e. there is no parent u that 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 7.3.

7.1.5 Monoid

A monoid is the same as a semiring, but it only has one operation which must be associative and 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 7.3.

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 7.3: A list of commonly used semirings and monoids in GraphBLAST.

7.1.6 Masking

Masking is an important tool in GraphBLAST that lets a user control for which indices they would like to see the result of any operation in Table 7.2 be written to the output. The indices they pass in is called the *mask* and must be in the form of a `Vector` or `Matrix` object. The masking semantic used is the following:

> *For 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: they want 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 does not have a 0, then it will not be written. This is called the *structural complement* of the mask.

7.1.7 Descriptor

A descriptor is an object passed into all operations listed in Table 7.2 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 7.4.

In our implementation, we choose not to include `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 so 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`.

We introduce an extension method `Descriptor::toggle(Desc_Field field)`. The semantic this method uses is that if the value for `field` is currently set to default, this

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

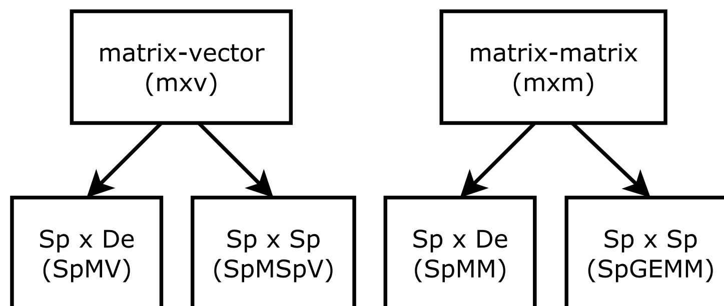


Figure 7.1: Decomposition of key GraphBLAS operations. Note that vxm is the same as mxv and setting the matrix to be transposed, so it is not shown.

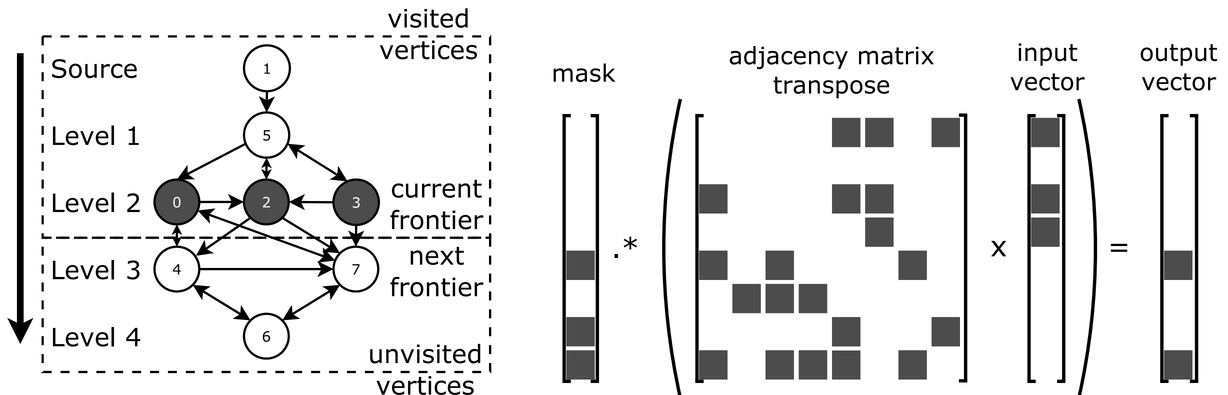
method will set it to the non-default value and if it is currently set to non-default, it will set it to the default value. We found that if we want to reuse codepaths in our backend e.g. for $\mathbf{A}^T\mathbf{f}$ and $\mathbf{fA}$, we can make the vector-matrix multiply call the matrix-vector codepath after calling `Descriptor::toggle(GrB_INP1)`.

Another useful case is found in our code example (see Algorithm 11 (right)). Here, we wanted to use the same Descriptor object for several methods that required different `GrB_MASK` settings. For example, the vector-matrix multiplication `vxm` requires the `GrB_SCMP` setting, but the `assign` requires the default setting. Instead of requiring the user to either: (1) use 2 Descriptor objects, or (2) use a getter method and have the user implement using if-else statements how they want to change the Descriptor object using `Descriptor::set`, we simplify the user experience by allowing them to call `Descriptor::toggle(GrB_MASK)`.

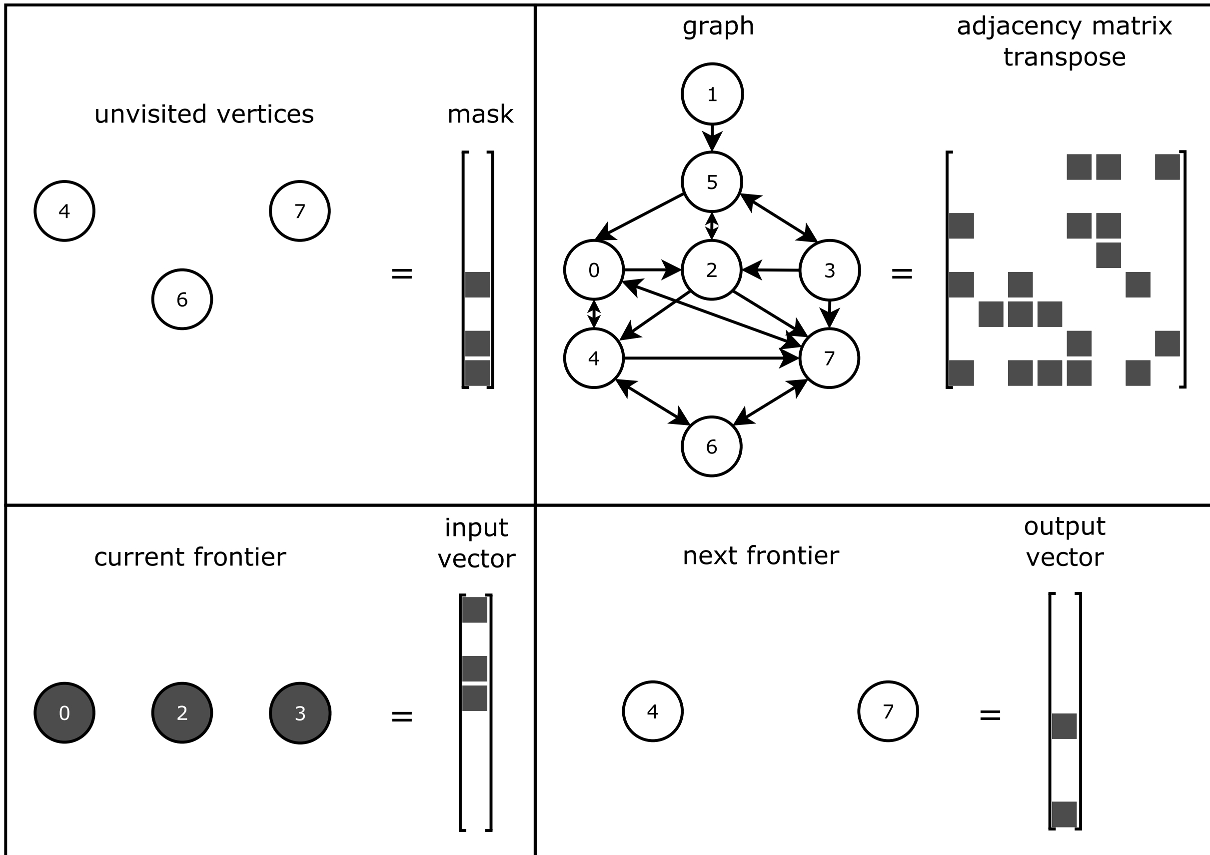
7.1.8 Key GraphBLAS operations

Looking at Table 7.2, it is clear the most computationally intensive operations are the `mxm`, `mxv`, and `vxm`. We find empirically that these operations take over 90% of application runtime. For these operations, we will decompose them into constituent parts in order to better optimize their performance (see Figure 7.1).

7.1.9 Running example



(a) Graph representation (b) Linear algebraic representation



(c) Graph concepts and linear algebraic equivalents

Figure 7.2: 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 current frontier (vertices 0, 2, 3) to their neighbors (vertices 4, 5, 7). This corresponds to the multiplication $\mathbf{A}^T\mathbf{f}$. This is followed by filtering out of visited vertices (vertex 5), leaving us with the next frontier (vertices 4, 7). This corresponds to the elementwise multiply $\neg\mathbf{v}. * (\mathbf{A}^T\mathbf{f})$.

As a running example in this paper, we will be talking about SpMV and SpMSpV with a direct dependence on graph traversal. The key step we will be discussing is Line 8 of Algorithm 11, which is the matrix-formulation of parallel breadth-first-search. Illustrated in Figure 7.2b, 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 $8 \times 8 = 64$ memory accesses. 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 memory accesses 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 we can use optimizations such as exploiting the input vector and output vector sparsity can further reduce the number of memory accesses required.

7.1.10 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 11. 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 they were first encountered d, (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 as c, (4) the iteration number is incremented. This while-loop continues until there are no more active vertices (c reaches 0).

Our code example differs from the GraphBLAS spec in the following ways:

1. We require `Matrix::build` and `Vector::build` to use `std::vector` rather than C-style arrays. However, we will maintain compatibility with GraphBLAS C API specification by allowing C-style arrays too.

2. We pass in a template parameter specifying the type in place of: (1) passing a datatype of `GrB_Type` to `Matrix` and `Vector` declaration, (2) specifying types used in the semiring.

3. We have predefined semirings and monoids, whose naming scheme follows that of C++ functors. As of May 2019, the latest version of GraphBLAS C API specification does not have predefined semirings so users must construct semirings themselves.

4. We have convenience methods `Vector::fill` and `Descriptor::toggle` that are not part of the GraphBLAS C API specification.

Regarding the use of template types, we plan to refactor our implementation to establish perfect compatibility with the GraphBLAS C API specification in the near future.

1: **procedure** MATRIXBFS(Graph $\mathbf{A}$, Vector $\mathbf{v}$, Source s)
2: Initialize $d \leftarrow 1$
3: Initialize $\mathbf{f}(i) \leftarrow \begin{cases} 1, & \text{if } i = s \\ 0, & \text{if } i \neq s \end{cases}$
4: Initialize $\mathbf{v} \leftarrow [0, 0, ..., 0]$
5: Initialize $c \leftarrow 1$
6: **while** $c > 0$ **do**
7: Update $\mathbf{v} \leftarrow d\mathbf{f} + \mathbf{v}$
8: Update $\mathbf{f} \leftarrow \mathbf{A}^{\mathbf{T}}\mathbf{f} .* \neg\mathbf{v}$ ▷ using Boolean semiring (see Table 7.3)
9: Compute $c \leftarrow \sum_{i=0}^{n} \mathbf{f}(i)$ ▷ using standard plus monoid (see Table 7.3)
10: Update $d \leftarrow d + 1$
11: **end while**
12: **end procedure**

```
1 #include <graphblas/graphblas.hpp>
2 using namespace graphblas;
3
4 void bfs(Vector<float>*       v,
5          const Matrix<float>* A,
6          Index                s,
7          Descriptor*          desc) {
8   Index A_nrows;
9   A->nrows(&A_nrows);
10  float d = 1.f;
11  Vector<float> f1(A_nrows);
12  Vector<float> f2(A_nrows);
13  std::vector<Index> indices(1, s);
14  std::vector<float> values(1, 1.f);
15  f1.build(&indices, &values, 1, GrB_NULL);
16  v->fill(0.f);
17  float c = 1.f;
18  while (c > 0) {
19    // Assign level d at indices f1 to visited vector v
20    assign(v, &f1, GrB_NULL, d, GrB_ALL, A_nrows, desc);
21    // Set mask to use structural complement (negation)
22    desc->toggle(GrB_MASK);
23    // Multiply frontier f1 by transpose of matrix A using visited vector
24    // v as mask
25    // Semiring: Boolean semiring (see Table 4)
26    vxm(&f2, v, GrB_NULL, LogicalOrAndSemiring<float>(), &f1, A, desc);
27    // Set mask to not use structural complement (negation)
28    desc->toggle(GrB_MASK);
29    f2.swap(&f1);
30    // Check how many vertices of frontier f1 are active, stop when
31    // number reaches 0
32    // Monoid: Standard addition (see Table 4)
33    reduce(&c, GrB_NULL, PlusMonoid<float>(), &f1, desc);
34    d++;
35  }
36 }
```

Algorithm 11: Matrix formulation of BFS (top) and example GraphBLAST code (bottom).

Major Feature	Component	Application BFS	SSSP	PR	TC
Exploit input sparsity	Generalized direction-optimization	✓	✓	✓	
	Boolean semiring	✓			
	Avoid sparse-to-dense conversion	✓	✓		
Exploit output sparsity	Masking	✓	✓		✓
Load-balancing	Static mapping	✓	✓	✓	✓
	Dynamic mapping (merge-based)	✓	✓	✓	

Table 7.5: Applicability of design principles.

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 hardware, GraphBLAS code can be used with minimal change. Currently, we are working on a Python frontend interface too, to allow users to build new graph algorithms without having to recompile. Over the next three sections, we will discuss the most important design principles for making this code performant, which are exploiting input sparsity, output sparsity and good load-balancing. Table 7.5 shows what applications our optimizations apply to.

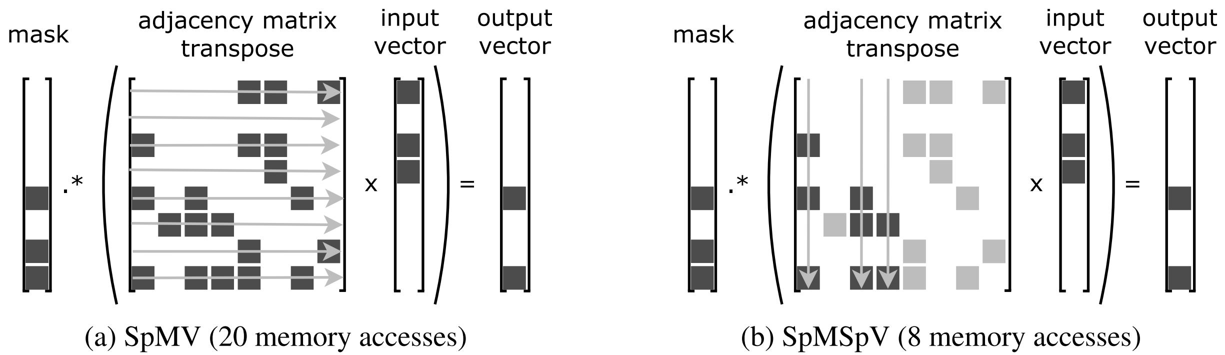
7.2 Exploiting Input Sparsity (Direction-Optimization)

In this section, we discuss our design philosophy of making exploiting input sparsity and one of its consequences, direction-optimization, a first-class citizen of our implementation. Since the matrix represents a graph, the matrix will be assumed to be stored in sparse format. Therefore, by *input sparsity*, we are referring to the input vector being sparse and exploiting 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 regardless of hardware. 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:

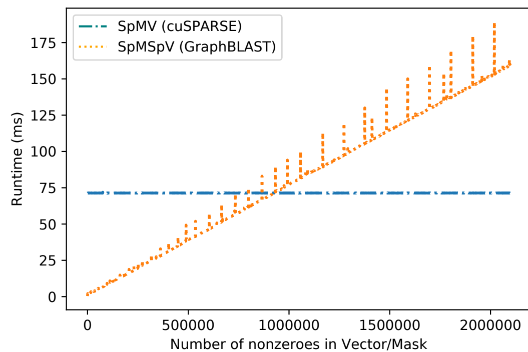
1. Previous work [10, 61] has shown that direction-optimization is critical to achieving state-of-the-art performance on breadth-first-search. However, direction-optimization has been notably absent in linear algebra-based graph frameworks and assumed only possible for traditional, vertex-centric graph 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 underspecified. As Algorithms 12 and 13 show, there are two ways to implement `mxv`. How should it be implemented?

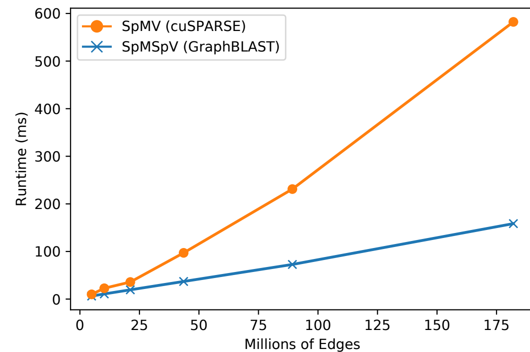


(a) SpMV (20 memory accesses)

(b) SpMSpV (8 memory accesses)



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



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

Figure 7.3: Comparison of SpMV and SpMSpV.

3. The GraphBLAS definition for `Matrix` and `Vector` objects are underspecified. What should the underlying data structure for these objects look like?

7.2.1 Two roads to matrix-vector multiplication

Before we address the above challenges, we will draw a distinction between two different ways the matrix-vector multiply $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ can be computed. We distinguish between SpMV (sparse matrix-dense vector multiplication) and SpMSpV (sparse matrix-sparse vector multiplication). There is extensive literature focusing on SpMV for GPUs (a survey can be found in [31]). 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 7.1.9 that by exploiting matrix sparsity (SpMV) in favor of dense matrix-vector multiplication (GEMV), we were able to bring the number of memory accesses down from GEMV's 64 memory accesses down to SpMV's 20. A natural question to ask is whether it is possible to decrease the number of memory accesses further when the input vector is sparse? Indeed when we exploit input sparsity (SpMSpV) to get the situation in Figure 7.3b, we can reduce the number of memory accesses from 20 down to 8. Similar how moving from GEMV to SpMV involved changing matrix storage format from dense to sparse, moving from SpMV to SpMSpV similarly involves 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 7.6.

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

Table 7.6: 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. Top three rows indicate the standard case $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$, while the bottom three rows represent the masked case $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x} .* \mathbf{m}$. Checkmarks indicate which form of sparsity each operation exploits.

Input: Sparse Matrix $\mathbf{A}$, Dense Vector $\mathbf{x}$
Output: Dense Vector $\mathbf{y}$

```
1: procedure SpMV(A, x, y)
2:     for i = 0 to A.nrows − 1 do
3:         y(i) ← A(i, :)x
4:     end for
5: end procedure
```

Algorithm 12: SpMV only exploits input matrix sparsity $\mathbf{A}(i,:)$.

Input: Sparse Matrix $\mathbf{A}$, Sparse Vector $\mathbf{x}$
Output: Sparse Vector $\mathbf{y}$

```
1: procedure SpMSpV(A, x, y)
2:     for i s.t. x(i) ≠ 0 do
3:         y ← A(:, i)x(i)
4:     end for
5: end procedure
```

Algorithm 13: SpMSpV exploits both input vector sparsity (i s.t. $\mathbf{x}(i) \neq 0$) and input matrix sparsity $\mathbf{A}(:,i)$.

7.2.2 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*. 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* refers to starting from the current frontier (set of vertices we are doing graph traversal from) and looking for children of this set of vertices. Then, out of this set of children, the previously visited vertices must be filtered out to generate the next iteration's frontier. On the other hand, *pull* refers to starting from the set of unvisited vertices and looking back to find their parents. Those nodes who have a parent in the current frontier, we will add them to next iteration's frontier. Beamer, Asanović and Patterson [10] observed that in the middle iterations of a BFS on scale-free graphs, the frontier would get large and each neighbor would be found many times leading to redundant work. In these iterations, they would switch to *pull* and 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-*

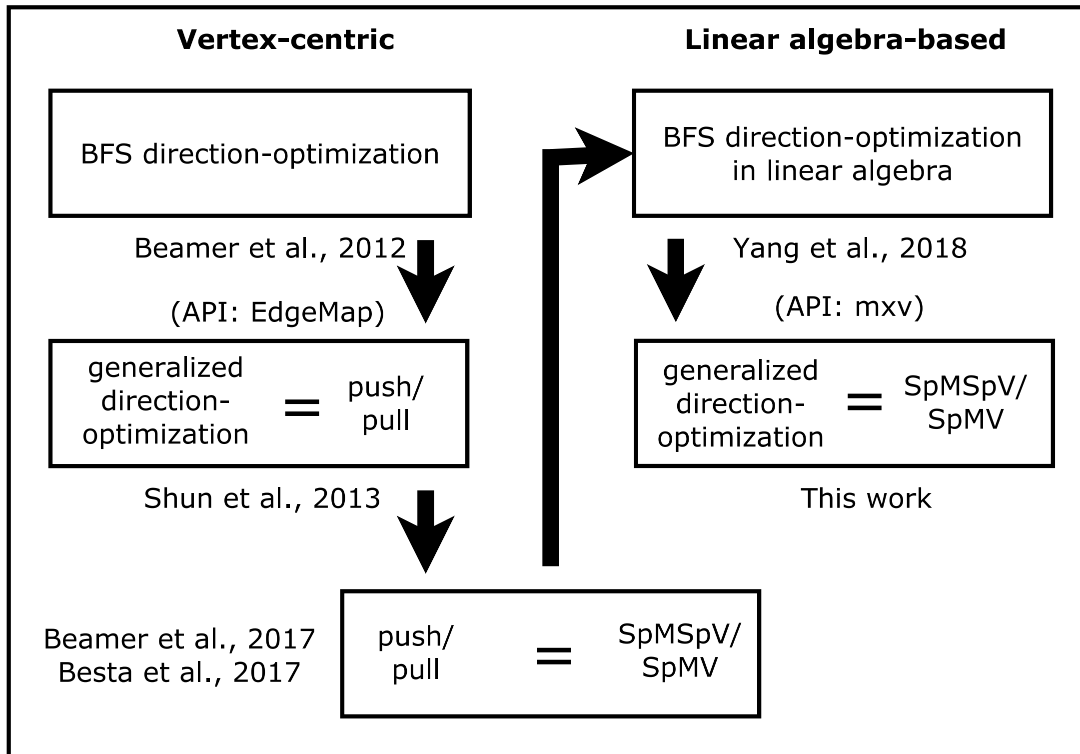


Figure 7.4: Where this work on direction-optimization fits in literature.

optimization or *direction-optimized BFS*, which was discovered by Beamer, Asanović and Patterson. Sometimes, it is also known as *push-pull*. Building on this work, Shun and Blelloch [61] were able to generalize direction-optimization to graph traversal algorithms beyond BFS. To avoid confusion with the BFS-specific instance, we refer to Shun and Blelloch's discovery *generalized direction-optimization*.

In Beamer, Asanović and Patterson later work [11], they looked at matrix-vector multiplication in the context of SpMV and SpMSpV-based implementations for PageRank. In both their work and that of Besta et al. [13], it was noted that switching between push/pull is the same as switching between SpMSpV/SpMV. In both works, authors show there is a one-to-one correspondence between push and SpMSpV, and between pull and SpMV; they are two ways of thinking about the same concept.

A summary of the work in this area is shown in Figure 7.4. Comparing with the work of Beamer, Asanović and Patterson and of Besta et al., our work differs in several ways: (1) they emphasize graph algorithm research, whereas we focus on building a graph framework, (2) their work targets multithreaded CPU, while ours targets GPU, and (3) their interface is vertex-centric, but ours is linear algebra-based.

Building on our earlier work, our contribution in this work is being the the first to extend the *generalized direction-optimization* technique to linear algebra-based framework based on the GraphBLAS specification. This is in contrast to previous implementations to the GraphBLAS specification, GBTL [73] and SuiteSparse [25], which do not support *generalized direction-optimization* and in consequence, trail the 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 [37], 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 as the vector sparsity increases. In other words, neither GBTL and SuiteSparse automatically switch to *pull* when the input vector becomes large in 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 (see Section 7.6).

7.2.3 Implementation

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

Direction-optimization Backend automatically handles direction-optimization when `mxv` is called, by calling either the SpMV or SpMSpV routine, whichever one yields fewer memory accesses based on an empirical cost model.

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 option to save memory by using a memory efficient, performance inefficient solution by only storing one of two representations. For `Vector`, since dense vector and sparse vector are required for the two different routines SpMV and SpMSpV respectively, give backend responsibility of switching between dense and sparse vector representations. Allow user to give hint as to the initial storage of the `Matrix` and `Vector` object.

7.2.3.1 Direction-optimization

Backend automatically handles direction-optimization when `mxv` is called, by calling either the SpMV or SpMSpV routine, whichever one yields fewer memory accesses based on an empirical cost model.

Table 7.7 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 $|E_f|$ requires prefix-sum computations. To avoid what in Beamer's paper 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 vector has around the same number of neighbors i.e. $d|V_f| \approx |E_f|$. Gunrock makes this assumption too.

2. When the mask (representing the unvisited vertices) is dense, counting the number of nonzeroes $|V_u|$ requires an additional GPU kernel launch, which represents significant overhead. Therefore, we make the assumption that the number of unvisited vertices is all

Work	Direction	Criteria	Application
Beamer et al. [10]	push $\rightarrow$ pull	$\|E_f\| > \|E_u\|/14$ and increasing	BFS only
	push $\leftarrow$ pull	$\|V_f\| < \|V\|/24$ and decreasing	BFS only
Ligra [61]	push $\rightarrow$ pull	$\|E_f\| > \|E\|/20$	generalized
	push $\leftarrow$ pull	$\|E_f\| < \|E\|/20$	generalized
Gunrock [67]	push $\rightarrow$ pull	$\|E_f^*\| > \|E_u^*\|/1000$	BFS only
	push $\leftarrow$ pull	$\|E_f^*\| < \|E_u^*\|/5$	BFS only
This work	push $\rightarrow$ pull	$\|E_f^*\| > \|E\|/10$	generalized
	push $\leftarrow$ pull	$\|E_f^*\| < \|E\|/10$	generalized

Table 7.7: Direction-optimization criteria for four different works. $|V_f|$ indicates number of nonzeroes in frontier $\mathbf{f}$. $|E_f|$ indicates number of neighbors from frontier $\mathbf{f}$. $|E_u|$ indicates number of neighbors from unvisited vertices. Superscript * indicates the value is approximated rather than precisely calculated.

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.

7.2.3.2 mxv: SpMV or SpMSpV

Following an earlier work by the authors [70], 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 how using `SparseVector` variant of `graphblas::mxv` performed compared with `DenseVector` as a function of `Vector` sparsity for a synthetic undirected Kronecker graph with 2M vertices and 182M edges. For more details of the experiment setup, see Section 7.6.

As our microbenchmark in Figure 7.3 illustrates, the SpMSpV variant of `graphblas::mxv` scales with the sparsity of the input vector. However, SpMV variant is constant. This matches the theoretical complexity shown in Table 7.6, which shows that SpMV scales with $\mathcal{O}(dM)$, which is irrespective 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. Another observation is that SpMSpV has an additional logarithmic factor compared to SpMV. This is because the columns must either be merged together in a multi-way merge, hash table or by using atomics.

7.2.3.3 Matrix and Vector storage format

One of the most important design choices an implementer needs to make 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 it exceeds available device memory. We use the popular CSR (Compressed Sparse Row), because they are standard in graph analytics and in order to support fast row access required by SpMV. Similarly since we need to support SpMSpV, the CSC data structure is necessary to support fast column access (see Figure 7.3).

For `Vector` objects, we have both a dense storage and a sparse storage. 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` as shown in Table 7.1, we allow the user to give the backend hints on whether they want the object to initially be stored in dense or sparse storage.

7.2.4 Direction-optimization insights

Exploiting input sparsity is a very useful and important strategy for high-performance in graph traversals. This section showed that the GraphBLAS interface is at the right level of abstraction that the user does not have to specify whether or not they want to exploit input sparsity; instead, they only need to write the code once using the `mxv` interface and both forms of SpMV and SpMSpV code are 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.

7.3 Exploiting Output Sparsity (Masking)

The previous section discussed how important it is to reduce the number of memory accesses by 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 up a matrix-vector or matrix-matrix multiply with an elementwise product. This gives the backend opportunity 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 citizen 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?

7.3.1 Motivation and applications of masking

Following the brief introduction to masking in Section 7.3, 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 7.2; 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 [18, 72] where 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 [24] where the mask `Vector` represents the set of vertices that have seen their distances from 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` that would otherwise be increasingly densifying sparse throughout the SSSP.

3. In adaptive PageRank (also known as PageRankDelta) [40, 61] where 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 [5, 69] where 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 $M \times M$ corresponds to finding for each index pair (i, j) the number of wedges $i \rightarrow k \rightarrow j$ that can be formed for all $k \in V$. Adding the mask Matrix then will yield $M \times M. * M$, which indicates the number of wedges that are also triangles by virtue of the presence of edge $i \rightarrow j$. The $.*$ operation indicates element-wise operation.

7.3.2 Microbenchmarks

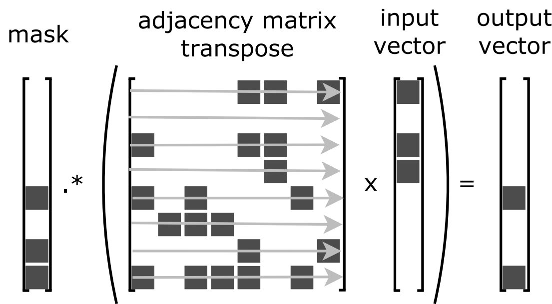
Similar to the earlier microbenchmark, we benchmark how using masked SpMV and SpMSpV variants of `graphblas::mxv` performed compared with unmasked SpMV and SpMSpV as a function of mask `Vector` sparsity for a synthetic undirected Kronecker graph with 2M vertices and 182M edges. For more details of the experiment setup, see Section 7.6.

As our microbenchmark in Figure 7.5 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 5.1, which shows that masked SpMV scales with $\mathcal{O}(d\, nnz(\mathbf{m}))$, where $\mathbf{m}$ is the mask `Vector`. However, 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.

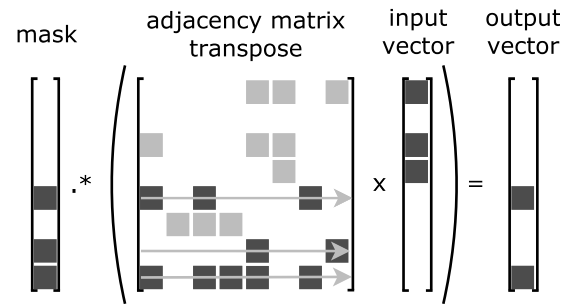
In the running example, recall in Figure 7.5a that standard SpMV, which performs the matrix-vector multiply followed by the elementwise product with the mask took 20 memory accesses. However, when we reverse the sequence of operations by loading the mask, seeing which elements of the mask are nonzero and then only doing the matrix-vector multiply for those rows that map to a nonzero mask element, we see that the number of memory accesses drops significantly from 20 down to 10.

7.4 Load-balancing

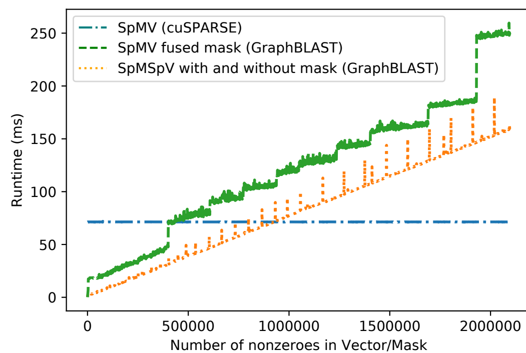
In this section, we discuss ways to implement the memory access patterns SpMV, SpMSpV, SpMM and SpGEMM in a way that tries to address the problem of *load imbalance*. We focus



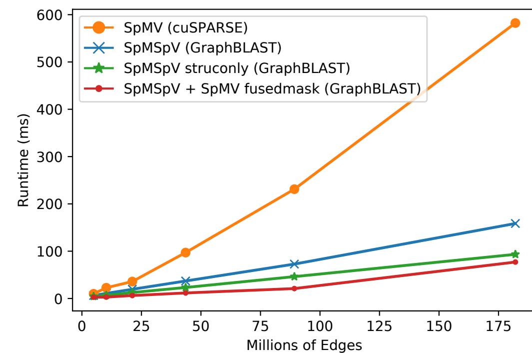
(a) SpMV not fused with mask (20 memory accesses)



(b) SpMV fused with mask (10 memory accesses)



(c) Algorithmic complexity of SpMSpV and SpMV with and without masking as a function of vector sparsity.



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

Figure 7.5: Comparison with and without fused mask.

on the backend implementations of these four patterns, because they are the implementations behind `mxm`, `mxv` and `vxm`, which are the most computationally intensive and important operations in GraphBLAS (see Table 7.2). These are the operations that make or break one's implementation of GraphBLAS.

Recall in Section 7.1. we mentioned that an operation is a memory access pattern common to many graph algorithms. A typical problem in graph algorithms is that most processing units have already completed their work, but a few are still working. This naturally emerges in algorithms such as breadth-first-search on scale-free graphs, where it is common for there to be a few supernodes with thousands of neighbors, yet most nodes with only tens of neighbors. In such a situation, if one were to simply assign one GPU warp (a computing unit composed of 32 threads) to each node in order to find its neighbors, the warps that got assigned these supernodes would become the bottleneck of the computation.

This is a case of *load imbalance*. We try to address this problem in the context of the four aforementioned memory access patterns. However, the problem or load imbalance cannot be solved completely. It often forces one to make tradeoffs in terms of the following:

- Graph traversal throughput (higher is better)
- Synchronizations (fewer is better)

Technique	Description
Static workload mapping	Assign thread, warp or CTA to matrix row or nonzero, but load balance could be arbitrarily bad depending on dataset
Dynamic workload mapping	Divide computations evenly amongst threads, but need to pay for load-balance overhead (usually 2 additional kernel launches and limited amount of global memory accesses)
Mask before multiply	Get benefit of mask sparsity, but may require more irregular memory access pattern such as binary search per thread
Mask after multiply	Miss out on benefit of mask sparsity, but no need to take on irregular memory access pattern

Table 7.8: Summary of load-balancing techniques used in GraphBLAST.

- Kernel launches (fewer is better)
- Memory accesses (fewer is better)

The load-balancing techniques we consider fall into two categories—workload mapping strategy and mask sparsity strategy (see Table 7.8). For masked variants of each memory access pattern, we must decide whether it is better to apply the mask (i.e. perform elementwise multiply) before the matrix multiply or after the matrix multiply.

7.4.1 Matrix-vector multiply

In matrix-vector multiply, the problem we are solving is $\mathbf{y} = \mathbf{A}\mathbf{x}$ and for the masked variant $\mathbf{y} = \mathbf{A}\mathbf{x}. * \mathbf{m}$.

7.4.1.1 SpMV: Sparse matrix-dense vector

For unmasked SpMV, the dynamic workload mapping, also known as *merge-based* load-balancing (load-balancing based on merge path [29, 35], which is an algorithm for merging two lists in parallel) was pioneered by Baxter [8] and Dalton, Olson and Bell [23]. However, Merrill and Garland [51] noted that this was not a true merge path-based solution. provided a dataset has an arbitrarily large number of empty rows. The difference between static and dynamic load-balancing schemes is shown in Figure 6.2.

Static workload mapping Assigns thread, warp or block to each matrix row.

Dynamic workload mapping We use the SpMV implementation from ModernGPU library [8] as our SpMV. We find that it outperforms static workload mapping in most circumstances that it can be used by default.

Masked variant Use static work mapping of assigning a warp per mask nonzero.

7.4.1.2 SpMSpV: Sparse matrix-sparse vector

Static workload mapping We are planning to explore this approach in the future.

Dynamic workload mapping We use the multi-kernel approach described by the authors in an earlier paper [70]. One advantage of this approach is that it allows the SpMSpV to be done without needing atomics. However, the disadvantage is that it requires $|E|$ additional memory. This approach is suitable for scale-free graphs, but has a lot of kernel launch and memory access overhead, so it does not do so well on road network graphs.

Masked variant The mask is better to be applied afterwards, because there are typically much more nonzeroes in the mask compared to the input vector.

7.4.2 Matrix-matrix multiply

In matrix-matrix multiply, the problem we are solving is $\mathbf{C} = \mathbf{AB}$ and for the masked variant $\mathbf{C} = \mathbf{AB}. * \mathbf{M}$.

7.4.2.1 SpMM: Sparse matrix-dense matrix

In this problem, we assume we are dealing with multiplying square sparse matrix by a tall-and-skinny dense matrix. This is due to the typical main memory of the GPU being limited to 12 GB. We make the following assumptions: 1) floating point precision, 2) the $1M \times 1M$ sparse matrix takes up 2 GB, and 3) the input and output being $1M \times N$ dense matrices taking 5 GB each. These assumptions yieldd a value of $N = 1250$ for the number of columns, which means in order to fit into GPU main memory the dense matrix must be a tall-and-skinny matrix.

Static workload mapping Assigns thread, warp or block to each matrix row.

Dynamic workload mapping We developed our own SpMM implementation [71] that is based on the dynamic mapping SpMV variant. We use this when the number of nonzeroes per row in $\mathbf{A}$ is less than 9.35.

Masked variant Typically, the mask can be assumed to be a sparse matrix, so the implementation follows masked SpGEMM. The only difference is the right-hand-side matrix (in this case, dense) can be directly indexed skipping the binary search.

7.4.2.2 SpGEMM: Sparse matrix-sparse matrix

Static workload mapping We base our hash table-based SpGEMM on work by Naumov et al. [55]. Following Naumov et al., we use a 2-step implementation of Gustavson's algorithm [37]: 1) In the first step, we count how many nonzeroes there will be in the output in order to populate the row pointers (deduplication done using the hash table, hash table size returns the number of nonzeroes), and 2) In the second step, we perform the multiplication. The memory access pattern in both steps is similar, and the only difference being the computation.

Dynamic workload mapping We are planning to explore this approach in the future.

Masked variant 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 static work mapping where we assign a warp per row of the mask $\mathbf{M}$, and for every nonzero $\mathbf{M}(i, j)$ in the mask each warp loads the row of $\mathbf{A}(i, :)$ in order

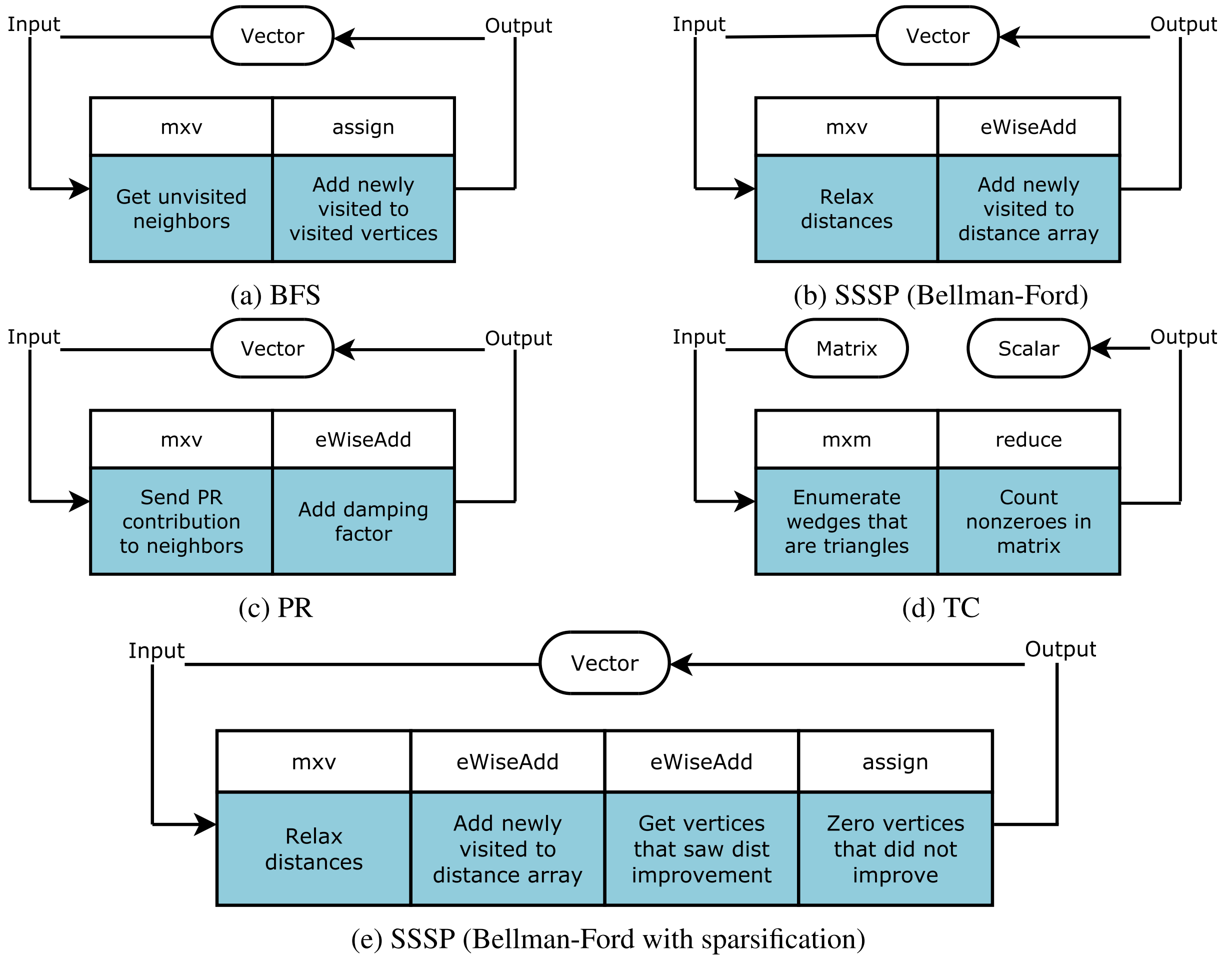


Figure 7.6: Operation flowchart for different algorithms expressed in GraphBLAS. A loop indicates a while-loop that runs until the Vector is empty.

to perform the dot-product $\mathbf{A}(i,:)\mathbf{B}(:,j)$. Using their $\mathbf{A}$-element, thread in the warp performs binary search on column $\mathbf{B}(:,j)$ and accumulates the result of the multiplication. After the row is finished, a warp reduction is done, and the output written to $\mathbf{C}(i,j)$.

7.5 Applications

One of the main advantages of GraphBLAS is that the operations can be composable to develop new graph algorithms. For each application in this section, we describe the hardwired GPU implementation and how our implementation can be expressed using GraphBLAS. Then the next section will compare performance between hardwired and GraphBLAS implementations. Figure 7.6 shows the GraphBLAS algorithms required to implement each algorithm

7.5.1 Breadth-first-search

Given a source vertex $s \in V$, a BFS is a full exploration of 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. [52] achieves its high performance through careful load-balancing, avoidance of atomics, and heuristics for avoiding redundant vertex discovery. Its chief operations are expand (to generate a new frontier) and contract (to remove redundant vertices) phases. Enterprise [47], a GPU-based BFS system, introduces a very efficient implementation that combines the benefits of direction optimization of Beamer, Asanović and Patterson [10], adaptive load-balancing workload mapping strategy of Merrill et al., and not synchronizing each BFS iteration which addresses the kernel launch overhead problem.

GraphBLAST implementation Merrill et al.'s expand and contract maps nicely to GraphBLAST's `mxv` operator with mask using a Boolean semiring. Like Enterprise, we implement efficient load-balancing (Section 7.4) and direction-optimization, which was described in greater detail in Section 7.2. We do not use Enterprise's method of skipping synchronization between BFS iterations, but we use optimizations *early-exit* and *structure-only*, that 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 authors in an earlier work [72].

7.5.2 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. [24]. 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 application code, and (2) sparsifying the set of active vertices after each iteration, by comparing each active vertex to see whether or not they improved over the stored distance in the distance array. The second step introduces two additional steps (compare Figures 7.6b and 7.6e).

7.5.3 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, 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 [40] (also known as PageRankDelta). In this paper, we do not implement or compare against this variant of PageRank.

Hardwired GPU Implementation One of the highest performing implementations of PageRank is written by Khorasani, Vora and Gupta [43]. In their system, they use solve the load imbalance and GPU underutilization problem with a GPU adoption of GraphChi's Parallel Sliding Window scheme [45]. They call this preprocessing step "G-Shard" and combine it with a concatenated window method to group edges from the same source IDs.

GraphBLAST implementation In GraphBLAST, we rely on the merge-based load-balancing scheme discussed in Section 7.4. 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 where the vector does not get any sparser, it is more efficient to use SpMV rather than SpMSpV.

7.5.4 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 coeffcient and transitivity ratio.

Hardwired GPU Implementation The best-performing implementation of triangle counting is by Bisson and Fatica [14]. There, they demonstrate the use of a static workload mapping of thread, warp, block per matrix row together with using bitmaps is a very good approach to this problem.

GraphBLAST Implementation In GraphBLAST, we follow Azad and Buluç [5] and Wolf et al. [69] 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 $\mathbf{L}$, it can be shown that the number of triangles is the reduction of the matrix $\mathbf{B} = \mathbf{L}\mathbf{L}^T .{*}\mathbf{L}$ to a scalar. 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 a straightforward static work mapping where we assign a warp per row of the mask $\mathbf{M}$, and for every nonzero $\mathbf{M}(i, j)$ in the mask each warp loads the row of $\mathbf{A}(i, :)$ in order to perform the dot-product $\mathbf{A}(i, :)\mathbf{B}(:, j)$. Using their $\mathbf{A}$-element, thread in the warp performs binary search on column $\mathbf{B}(:, j)$ and

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

Table 7.9: 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 above the middle divide are used for BFS, SSSP and PR. Datasets below it are used for TC. An asterisk indicates the diameter is estimated using samples from 10,000 vertices.

accumulates the result of the multiplication. After the row is finished, a warp reduction is done, and the output written to $\mathbf{C}(i, j)$. This is followed by a reduction of matrix $\mathbf{C}$ to a scalar returning the number of triangles in graph $\mathbf{A}$.

7.6 Experimental Results

We first show overall performance analysis of GraphBLAST on nine datasets including both real-world and generated graphs; the topology of these datasets spans from regular to scale-free. Five additional datasets are used specifically for triangle counting, because they are the ones typically used for comparison of triangle counting [14, 66].

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. Ligra was compiled using icpc 15.0.1 with CilkPlus. SuiteSparse was compiled using g++ 4.9.3. 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.

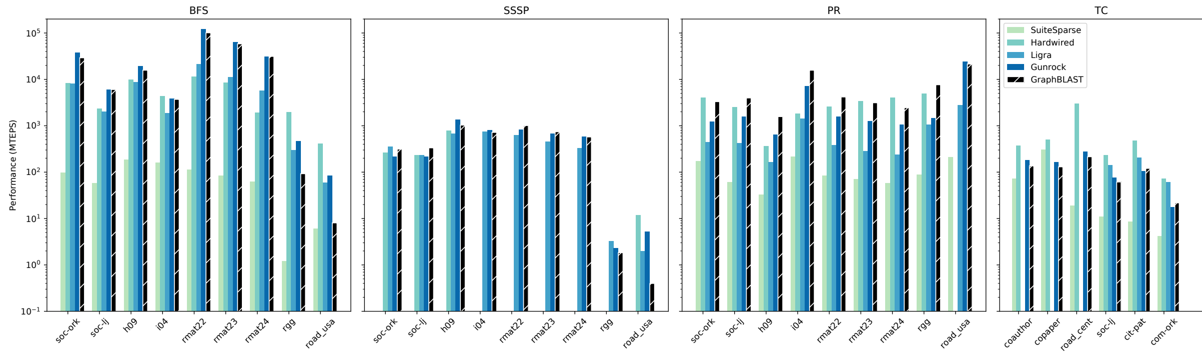
Datasets We summarize the datasets in Table 7.9. soc-orkut (soc-ork), com-Orkut (com-ork), soc-Livejournal1 (soc-lj), hollywood-09 (h09) are social graphs; indochina-04 (i04) is a crawled hyperlink graph from indochina web domains; coAuthorsCiteseer (coauthor), coPapersDBLP (copaper), and cit-Patents (cit-pat) are academic citation and patent citation networks; rmat_s22_e64 (rmat-22), rmat_s23_e32 (rmat-23), and rmat_s24_e16 (rmat-24) are three

Alg.	Dataset	Runtime (ms) [lower is better]					Edge throughput (MTEPS) [higher is better]				
		SuiteSparse GraphBLAS	Hardwired GPU	Ligra	Gunrock	GraphBLAST	SuiteSparse	Hardwired GPU	Ligra	Gunrock	GraphBLAST
BFS	soc-ork	2165	25.81	26.1	**5.573**	7.230	98.24	12360	8149	**38165**	29217
	soc-lj	1483	36.29	42.4	**14.05**	14.16	57.76	5661	2021	**6097**	6049
	h09	596.7	11.37	12.8	**5.835**	7.138	188.7	14866	8798	**19299**	15775
	i04	1866	**67.7**	157	77.21	80.37	159.8	**8491**	1899	3861	3709
	rmat-22	4226	41.81	22.6	**3.943**	4.781	114.3	17930	21374	**122516**	101038
	rmat-23	6033	59.71	45.6	**7.997**	8.655	83.81	12971	11089	**63227**	58417
	rmat-24	8193	270.6	89.6	16.74	**16.59**	63.42	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	NI	807.2	**595**	981.6	676.7	NI	263.5	**357.5**	216.7	314.3
	soc-lj	NI	369	368	393.2	**256.3**	NI	232.2	232.8	217.9	**334.2**
	h09	NI	143.8	164	**83.2**	109.123	NI	783.4	686.9	**1354**	1032
	i04	NI	—	397	**371.8**	414.5	NI	—	750.8	**801.7**	719.2
	rmat-22	NI	—	774	583.9	**477.5**	NI	—	624.1	827.3	**1011.7**
	rmat-23	NI	—	1110	739.1	**680.0**	NI	—	455.5	684.1	**743.6**
	rmat-24	NI	—	1560	**884.5**	905.2	NI	—	333.1	**587.5**	574.0
	rgg	NI	—	**80800**	115554	144291	NI	—	**3.28**	2.294	1.84
	road_usa	NI	**4860**	29200	11037	144962	NI	**11.87**	1.98	5.229	0.398
PageRank	soc-ork	1230	**52.54**	476	173.1	64.22	173.0	**4048**	446.8	1229	3312
	soc-lj	1386	33.61	200	54.1	**21.54**	61.83	2550	428.5	1584	**3978**
	h09	386.8	34.71	77.4	20.05	**8.12**	33.10	368.8	165.4	638.4	**1577**
	i04	1390	164.6	210	41.59	**19.16**	217.3	1835	1438	7261	**15763**
	rmat-22	5764	188.5	1250	304.5	**115.6**	83.79	2562	386.4	1586	**4178**
	rmat-23	7089	**147**	1770	397.2	161.3	71.32	**3439**	285.6	1273	3134
	rmat-24	8895	**128**	2180	493.2	211.5	58.42	**4060**	238.4	1054	2457
	rgg	2991	53.93	247	181.3	**34.58**	88.64	4916	1073	1462	**7665**
	road_usa	2746	—	209	**24.11**	26.91	210.2	—	2761	**23936**	21449
TC	coauthor	11.06	**2.2**	—	4.51	5.96	73.6	**370**	—	181	137
	copaper	103.8	**64.4**	—	197	246	309	**498**	—	163	130
	soc-lj	6322	**295**	490	896	1125	10.9	**234**	141	77.0	61.3
	cit-pat	1907	**34.5**	79.5	156	137	8.65	**478**	208	105	121
	com-ork	27887	**1626**	1920	6636	5367	4.2	**72.1**	61.0	17.7	21.8
	road_cent	895.2	**5.6**	—	61.4	78.7	18.9	**3018**	—	275	215

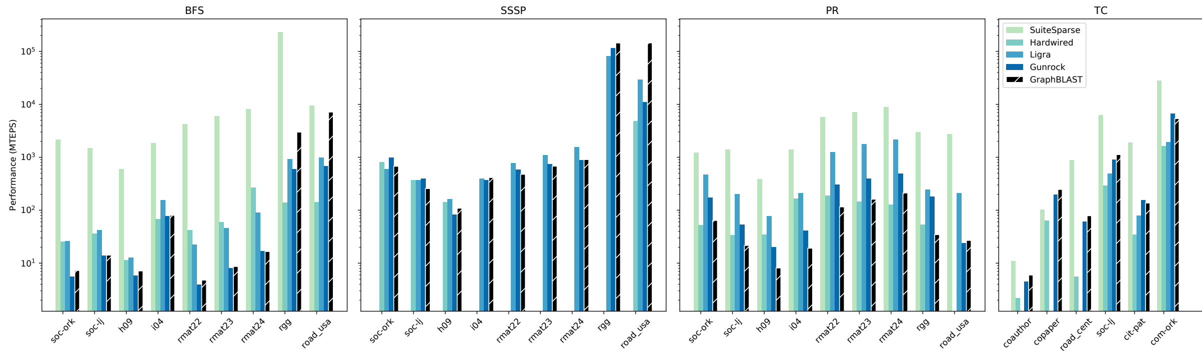
Table 7.10: GraphBLAST's performance comparison (runtime and edge throughput) with other graph libraries (SuiteSparse, Ligra, Gunrock) and hardwired GPU implementations on a Tesla K40c GPU. All PageRank times are normalized to one iteration. Hardwired GPU implementations for each primitive are Enterprise (BFS) [47], delta-stepping SSSP [24], pull-based PR [43], and triangle counting [14]. NI means the algorithm is not implemented on a framework. A missing data entry means either there is a runtime error.

generated R-MAT graphs. All seven datasets are scale-free graphs with diameters of less than 30 and unevenly distributed node degrees (80% of nodes have degree less than 64). Both rgg_n_24 (rgg), road_central (road_cent) and roadnet_USA (road_usa) datasets have large diameters with small and evenly distributed node degrees (most nodes have degree less than 12). soc-ork and com-Ork are from Network Repository [60]; soc-lj, i04, h09, road_central, road_usa, coauthor, copaper, and cit-pat are from University of Florida Sparse Matrix Collection [26]; rmat-22, rmat-23, rmat-24, and rgg are R-MAT and random geometric graphs we generated. The edge weight values (used in SSSP) for each dataset are random values between 1 and 64.

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 MTEPS is measured by recording the number of edges visited during the running (the sum of neighbor list lengths of all visited vertices) divided by the runtime. When a library does not report MTEPS, we use the following equation to compute it: $\frac{|E|}{t}$ where E is the number of edges in the graph and t is runtime.



(a) Performance (MTEPS) [higher is better]



(b) Runtime (ms) [lower is better]

Figure 7.7: Performance in MTEPS and milliseconds for GraphBLAST vs. four other graph processing libraries and hardwired algorithms on nine different graph inputs. Data is from Table 7.10.

7.6.1 Performance summary

Table 7.10 and Figure 7.7 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 irregular graphs. Therefore, we incur certain amount of overhead for our merge-based load-balancing and requiring kernel launch in every iteration. For these types of graphs, asynchronous approaches pioneered by Enterprise [47] 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 remains 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.

7.6.2 Comparison with CPU graph frameworks

We compare GraphBLAST's performance with two CPU graph libraries: the SuiteSparse GraphBLAS library, the first GraphBLAS implementation for single-threaded CPU [25]; and Ligra [61],

Algorithm Frontend Language	Ligra [61] C	Gunrock [67] C++	This work C++
Breadth-first-search	29	2732	25
Single-source shortest-path	55	857	25
PageRank	74	2006	27
Triangle counting	55	555	6

Table 7.11: Comparison of lines of C++ application code for three graph frameworks.

one of the highest-performing multi-core shared-memory graph libraries. Against SuiteSparse, the speedup of GraphBLAST on average on all algorithms is geomean $36\times$ ($892\times$ peak). Compared to Ligra, GraphBLAST's performance is generally comparable on most tested graph algorithms; note Ligra results are on a 2-CPU machine. We are $3.38\times$ ($1.35\times$ 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 7.5. However, we are $4.88\times$ slower on the road network graphs. For SSSP, a similar picture emerges. Compared to Ligra for scale-free graphs, our Bellman-Ford with sparsification algorithm with Ligra's Bellman-Ford algorithm means we get $1.35\times$ ($1.72\times$ peak) speed-up, but are $2.98\times$ slower on the road networks. For PR, we are $9.23\times$ ($10.96\times$ peak) faster, because we use a highly-optimized merge-based load balancer that is suitable for this SpMV-based problem. With regards to TC, we are $2.80\times$ slower, because we have a simple algorithm for the masked matrix-matrix multiply.

7.6.3 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 with . For TC, GraphBLAST is 3.3x 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 Gunrock, the fastest GPU graph framework, GraphBLAST's performance is comparable on BFS and TC with Gunrock being 11.8% and 11.1% faster respectively in the geomean. On SSSP, GraphBLAST is faster by $1.1\times$ ($1.53\times$ 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 $2.39\times$ ($5.24\times$ peak). For PR, the speed-up 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.

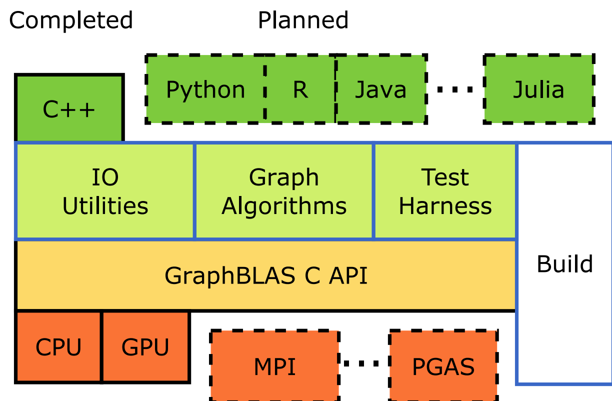


Figure 7.8: Design of GraphBLAST: completed and planned components.

In addition to getting comparable or faster performance, GraphBLAST has the advantage of being concise as shown by Table 7.11. 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 7.8). 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 know-how.

Chapter 8

Conclusion

In this paper, we set out to answer the question: What is needed for a high-performance graph framework that is based in linear algebra? The answer we propose is that it must: (1) exploit input sparsity through direction-optimization, (2) exploit output sparsity through masking, and (3) have a good load-balancing scheme. 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 four graph algorithms, we were able to obtain $36\times$ geomean $892\times$ peak) over SuiteSparse GraphBLAS (sequential CPU) and $2.14\times$ geomean ($10.97\times$ peak) and $1.01\times$ ($5.24\times$ peak) speed-up over Ligra and Gunrock respectively, which are state-of-the-art graph frameworks on CPU and GPU.

By construction, the GraphBLAS open standard establishes its first two goals—*portable performance* and *conciseness*—the former by making implementers adhere to the same standard interface, and the latter by basing the interface design around the language of mathematics, which is the most concise form of expression known to man. In this paper, we set out to meet the third goal of *high-performance*, which is the first step towards the fourth goal of *scalability*. Having established that GraphBLAS is capable of effectiveness at the small scale, it remains for researchers to determine whether it is also effective at the exascale. It is possible that expressing the problem as matrix multiplication can more easily allow researchers to handle the graph partioning in a rigorous rather than ad hoc fashion.

8.1 Future Directions

Scalability By construction, the GraphBLAS open standard establishes its first two goals—*portable performance* and *conciseness*—the former by making implementers adhere to the same standard interface, and the latter 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 the first step towards the fourth goal of *scalability*. Having established that GraphBLAS is capable of effectiveness at a single GPU scale, it remains for researchers to determine whether it is also effective at the exascale. It is possible that expressing the problem as matrix multiplication can more easily allow researchers to handle the graph partioning in a rigorous rather than ad hoc fashion.

GPU-based implementations have typically found difficulty in scaling to as many nodes as

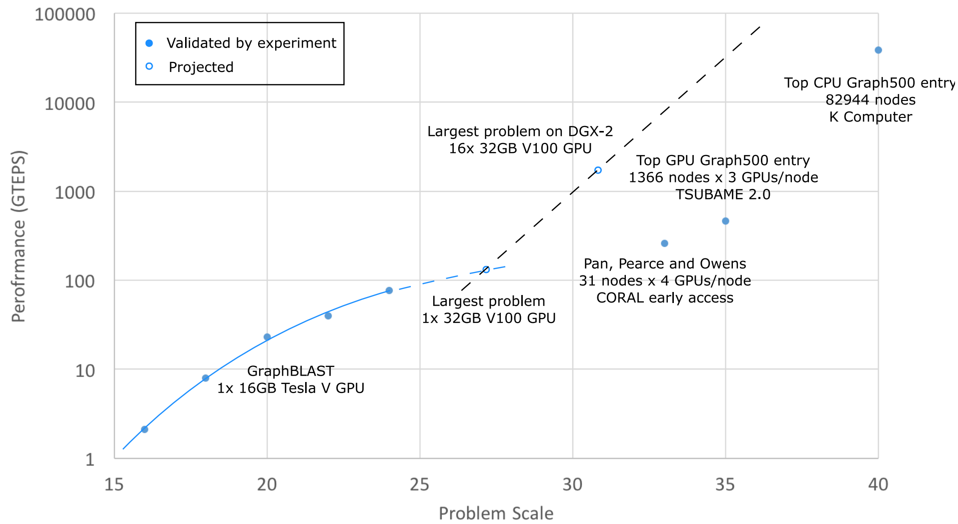


Figure 8.1: Data points from GraphBLAST and points representative of state-of-the-art in distributed BFS. Dashed line indicates projected performance assuming perfect scaling from 1 GPU. In random graph generation for each problem scale $SCALE$, the graph will have 2^{SCALE} vertices and 16×2^{SCALE} edges according to Graph500 rules.

CPU-based implementations, partly due to GPUs making the compute take less time, thus being sensitive to waiting any amount of time for inter-node communication and 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 $16\times$ GPUs with 32GB memory each and by being connected using NVSwitch technology that offers a bisection bandwidth of 2.4TB/s, the DGX-2 may be a contender for multiple GPU top BFS performance. For example in Figure 8.1, the dashed line and hollow point indicates the performance of DGX-2 system assuming linear scalability from the $1\times$ GPU GraphBLAST BFS would exceed current GPU leaders on Graph-500.

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. In the present situation, a GraphBLAS-based triangle counting algorithm 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, and just-in-time (JIT) compile the code to machine code, and execute the fused kernel.

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

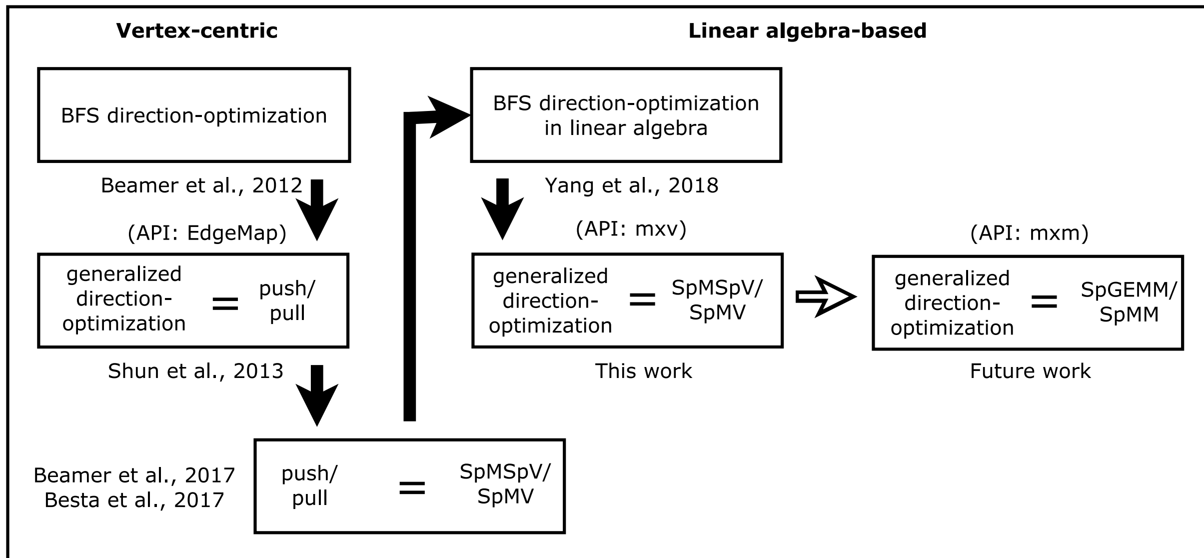


Figure 8.2: Another possible generalization of direction-optimization.

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 [47] 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 may turn out to be straightforward to solve, but the latter problem may also have implications when scaling to distributed implementations.

Matrix-matrix generalization of direction-optimization Currently, direction-optimization is only active 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 a 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). This could be abstracted away from the user as shown in Figure 8.2.

Memory Allocator Memory on the GPU is a precious resource compared to CPUs. On GPU operation timescales, memory allocation and deallocation can take significant portion of runtime. Three approaches are common:

1. Simple policy: This policy is to allocate and free GPU memory for each core operation. This is the simplest method to implement, but may incur significant overhead for compute-unintensive tasks. This method is used by the linear algebra library MAGMA [63].

2. Sophisticated policy: This policy will request all GPU memory at start-up. A sophisticated runtime then serves and recollects memory to and from data structures as necessary. This method is used by the deep learning library TensorFlow [1].

3. Greedy policy: This policy allocates temporary GPU memory required to do each core operation, and stores it in a buffer. This memory is not recollected after each operation. The memory is only freed when required temporary memory for a certain operation exceeds what it has in the buffer, and a larger buffer size is allocated. The disadvantage is this is not good software engineering practice, because it does not result in a clean separation of concerns. The backend developer must keep in mind how to split up a single buffer into multiple temporary arrays. This policy is what is currently implemented, and it can be thought of as a middle-of-the-road approach in between the simple and sophisticated policies.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[2] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing (HPC)*, HPC '15, pages 75–82, 2015.

[3] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal of Scientific Computing*, 38(6):C624–C651, 2016.

[4] Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. IPDPS '17, pages 688–697. IEEE, 2017.

[5] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 804–811. IEEE, 2015.

[6] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

[7] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, PPoPP '12, pages 23–34, February 2012. ISBN 978-1-4503-1160-1. doi: `10.1145/2145816.2145820`.

[8] Sean Baxter. Modern GPU Library. `http://nvlabs.github.io/moderngpu/`, 2015. Accessed: 2015-02-22.

[9] Scott Beamer. *Understanding and improving graph algorithm performance*. PhD thesis, University of California, Berkeley, 2016.

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

[11] Scott Beamer, Krste Asanović, and David Patterson. Reducing Pagerank communication via propagation blocking. IPDPS '17, pages 820–831. IEEE, May 2017.

[12] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 18:1–18:11, Nov. 2009. doi: `10.1145/1654059.1654078`.

[13] 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 International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, HPDC '17, pages 93–104. ACM, 2017.

[14] Mauro Bisson and Massimiliano Fatica. High performance exact triangle counting on

GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(12):3501–3510, 2017.

[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, 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)*, SC '11, pages 65:1–65:12, November 2011.

[17] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the GraphBLAS API for C. Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 643–652. IEEE, 2017.

[18] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. *The GraphBLAS C API Specification*, 11 2017. Rev. 1.1.

[19] Aydın Buluç and John R. Gilbert. On the representation and multiplication of hypersparse matrices. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '08, April 2008.

[20] S. Changpinyo, M. Sandler, and A. Zhmoginov. The power of sparsity in convolutional neural networks. *arXiv preprint arXiv:1702.06257*, 2017.

[21] Avery Ching. Giraph: Large-scale graph processing infrastructure on Hadoop. *Proceedings of the Hadoop Summit, Santa Clara*, 11(3):5–9, 2011.

[22] 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, 2015.

[23] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix-matrix multiplication for the GPU. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25, 2015.

[24] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '14, pages 349–359, May 2014. doi: `10.1109/IPDPS.2014.45`.

[25] Tim Davis. SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 2018. Accessed: 2019-05-01.

[26] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[27] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[28] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139. Springer, 2009.

[29] Narsingh Deo, Amit Jain, and Muralidhar Medidi. An optimal parallel algorithm for merging using multiselection. 1994.

[30] Joe Eaton. nvgraph. `https://docs.nvidia.com/cuda/nvgraph/index.html`, 2016. Accessed: 2018-01-18.

[31] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):30, 2017.

[32] 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 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 10. ACM/IEEE, 2018. URL `http://eecs.berkeley.edu/~aydin/a10-georganas.pdf`.

[33] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High-performance graph algorithms from parallel sparse matrices. In *Applied Parallel Computing: State of the Art in Scientific Computing*, volume 4699 of *LNCS*, pages 260–269. Springer, March 2007.

[34] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 17–30. USENIX Association, October 2012.

[35] Oded Green, Robert McColl, and David A Bader. GPU merge path: A GPU merging algorithm. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, ICS '12, pages 331–340. ACM, 2012.

[36] Gero Greiner and Riko Jacob. The I/O complexity of sparse matrix dense matrix multiplication. In *LATIN*, pages 143–156, 2010.

[37] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[38] James A Jablin, Thomas B Jablin, Onur Mutlu, and Maurice Herlihy. Warp-aware trace scheduling for GPUs. In *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, PACT '14, pages 163–174, 2014.

[39] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. pages 2328–2337, 2018.

[40] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. Adaptive methods for the computation of pagerank. *Linear Algebra and its Applications*, 386:51–65, 2004.

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

[42] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[43] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '14, pages 239–252. ACM, 2014.

[44] Denes Konig. Graphen und matrizen (Graphs and matrices). *Matematikai Lapok*, 38: 116–119, 1931.

[45] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.`

cfm?id=2387880.2387884.

[46] Nikolaj Leischner. *GPU algorithms for comparison-based sorting and merging based on multiway selection*. PhD thesis, Karlsruhe Institute of Technology, 2010.

[47] 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)*, SC '15, pages 68:1–68:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807594.

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

[49] Timothy G Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E Moreira. GraphBLAS C API: Ideas for future versions of the specification. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, 2017. URL http://eecs.berkeley.edu/~aydin/GrB_futures_hpec17.pdf.

[50] Duane Merrill. CUB Library. http://nvlabs.github.io/cub, 2015. Accessed: 2015-02-22.

[51] 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)*, SC '16, pages 678–689. IEEE, Nov. 2016.

[52] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

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

[54] M Naumov, LS Chien, P Vandermersch, and U Kapasi. CUSPARSE Library: A set of basic linear algebra subroutines for sparse matrices. In *GPU Technology Conference*, volume 2070, 2010.

[55] M Naumov, M Arsaev, P Castonguay, J Cohen, J Demouth, J Eaton, S Layton, N Markovskiy, I Reguly, Nikolai Sakharnykh, et al. AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015.

[56] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. FastSpMM: An efficient library for sparse matrix matrix product on GPUs. *The Computer Journal*, 57 (7):968–979, 2013.

[57] 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 '19, May 2019.

[58] John D. Owens. Hive applications. https://gunrock.github.io/docs/hive_year1_summary.html, 2018.

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

[60] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL http://networkrepository.com.

[61] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and*

Practice of Parallel Programming (PPoPP), PPoPP '13, pages 135–146, New York, NY, USA, February 2013. ACM. ISBN 978-1-4503-1922-5.

[62] 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, 2015.

[63] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010. ISSN 0167-8191. doi: 10.1016/j.parco.2009.12.005.

[64] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC '08, pages 31:1–31:11, Nov. 2008. doi: 10.1145/1413370.1413402.

[65] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Fall 2003.

[66] 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 ACM Workshop on High Performance Graph Processing (HPGP)*, pages 1–8. ACM, 2016.

[67] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):3, 2017.

[68] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785.

[69] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[70] Carl Yang, Yangzihao Wang, and John D Owens. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 841–847. IEEE, 2015.

[71] Carl Yang, Aydın Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the GPU. August 2018. doi: 10.1007/978-3-319-96983-1_48. URL https://escholarship.org/uc/item/5h35w3b7.

[72] Carl Yang, Aydın Buluç, and John D. Owens. Implementing push-pull efficiently in GraphBLAS. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, ICPP 2018, pages 89:1–89:11, August 2018. doi: 10.1145/3225058.3225122.

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