To tile or not to tile, that is the question

Altan Haan
Doru Thom Popovici
Koushik Sen
Costin Iancu
Alvin Cheung

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-11
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-11.html

April 3, 2025

To tile or not to tile, that is the question

Altan Haan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

Professor Alvin Cheung
Research Advisor

April 3, 2025

(Date)

* * *

Professor Max Willsey
Second Reader

April 3, 2025

(Date)

Abstract

A wide variety of graph algorithms expressed as linear algebra operations, i.e., triangle counting, k-truss analysis, breath first search, betweenness centrality, depend on the masked sparse matrix time sparse matrix multiplication (masked-SpGEMM) kernel. SuiteSparse:GraphBLAS, the de-facto sparse linear algebra library for graph analytics, offers support for this specific computation. Under a simple API, the library offers multiple masked-SpGEMM implementations. While attempting to provide a flexible solution that adapts to the input graphs/data, the system uses heuristics to choose between implementations. It hides the mechanisms behind layers of complex code, making it hard for users to reason about performance. In this work, we provide an in-depth analysis of the design choices that affect the performance of the masked-SpGEMM, using triangle counting as the benchmark. We look at 1) techniques for load balancing the sparse computation across multiple threads, 2) the iteration space for traversing the matrix multiplication and masking operation, and 3) the implementation of the accumulator used to store the intermediate results. We discuss the trade-offs and show a detailed performance analysis of the implementations on shared memory systems for a wide variety of input graphs, comparing SuiteSparse:GraphBLAS and a highly optimized masked-SpGEMM implementation, and discuss future research directions given our observations.

To tile or not to tile, that is the question

Altan Haan*, Doru Thom Popovici†, Koushik Sen*, Costin Iancu†, Alvin Cheung*
*University of California, Berkeley †Lawrence Berkeley National Lab
altanh@cs.berkeley.edu, dtpopovici@lbl.gov, ksen@cs.berkeley.edu, cciancu@lbl.gov, akcheung@cs.berkeley.edu

***Abstract*—A wide variety of graph algorithms expressed as linear algebra operations, i.e., triangle counting, k-truss analysis, breath first search, betweenness centrality, depend on the masked sparse matrix time sparse matrix multiplication (masked-SpGEMM) kernel. SuiteSparse:GraphBLAS, the de-facto sparse linear algebra library for graph analytics, offers support for this specific computation. Under a simple API, the library offers multiple masked-SpGEMM implementations. While attempting to provide a flexible solution that adapts to the input graphs/data, the system uses heuristics to choose between implementations. It hides the mechanisms behind layers of complex code, making it hard for users to reason about performance. In this work, we provide an in-depth analysis of the design choices that affect the performance of the masked-SpGEMM, using triangle counting as the benchmark. We look at 1) techniques for load balancing the sparse computation across multiple threads, 2) the iteration space for traversing the matrix multiplication and masking operation, and 3) the implementation of the accumulator used to store the intermediate results. We discuss the trade-offs and show a detailed performance analysis of the implementations on shared memory systems for a wide variety of input graphs, comparing SuiteSparse:GraphBLAS and a highly optimized masked-SpGEMM implementation, and discuss future research directions given our observations.**

***Index Terms*—Graph Analytics, Load Balancing, Sparse Accumulators, Predictable Performance**

I. Introduction

The development of efficient graph analytics frameworks and algorithms is crucial due to the ever increasing size of data produced in scientific fields like social networks, recommender systems and bio-informatics. Over the past years, there have been a multitude of projects [1]–[6] focused on fast processing of sparse data represented as graphs. One such project is GraphBLAS [7] and SuiteSparse:GraphBLAS [8], an API and framework focused on expressing graph algorithms as sparse linear algebra operations respectively, analogous to the (dense) Basic Linear Algebra Subroutines (BLAS). Typically, graphs are represented using adjacency matrices, where the elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. GraphBLAS proposes graph algorithms to be expressed as operations applied on the adjacency matrix, with steps that can be composed to form larger algorithms such as triangle counting [9]–[11], k-truss calculation [12]–[14], breath first search [15], and betweenness centrality [16].

The masked sparse-matrix-times-sparse-matrix multiplication (masked-SpGEMM) operation is one such operation that is central to multiple graph algorithms. For example, to count the number of triangles (i.e., three interconnected nodes), one can multiply the adjacency matrix with itself to determine the paths of length two between all nodes, and then filter the result by requiring an extra path of length one between the corresponding nodes. The filtering step is performed by masking the intermediate result with the original adjacency matrix. In practice, this set of operations is performed in one step. In addition SuiteSparse:GraphBLAS chooses heuristically the implementation that best suits the input graph. The library attempts to offer a solution that can adapt to the diverse set of graph data structures and inputs. Figure 1 shows the execution time for the masked-SpGEMM for different input graphs, using GraphBLAS (or more specifically SuiteSparse:GraphBLAS) and GrB [17]. The execution time of the two implementations vary for the different input graphs. However, given the complexity of the SuiteSparse:GraphBLAS library, it can be hard to reason about its performance.

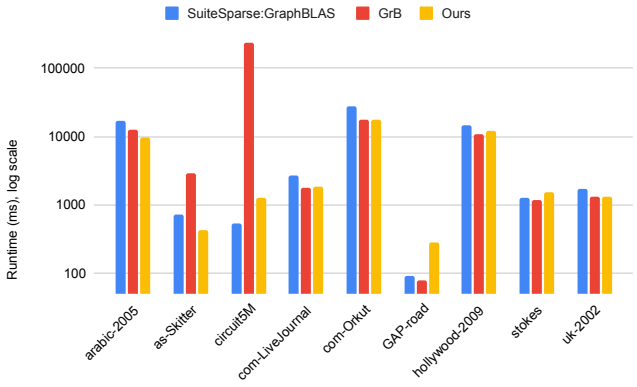


Fig. 1: Log scale execution times for the masked-SpGEMM using SuiteSparse:GraphBLAS [8], GrB [17], and our tuned implementation for a number of input graphs. All runs use a hash-based accumulator and are parallelized on an AMD CPU using 64 threads. While for some graphs the implementations are similar, there are outliers where SuiteSparse:GraphBLAS under-performs compared GrB, and vice-versa. Our tuned implementation eliminates most extreme outliers from GrB but still occasionally underperforms.

In this work, we perform an in-depth analysis on the trade-offs needed to achieve efficient implementations for the masked-SpGEMM kernel. We investigate three aspects of the computation. First, we look at tiling the computation and distributing the tiles across the threads to achieve a balanced execution. The work among threads needs to be balanced, but also the amount of data read from main memory needs to be reduced. Second, we focus on the iteration space for traversing the matrix multiplication and masking operation in one single step. We outline that there are multiple approaches to apply

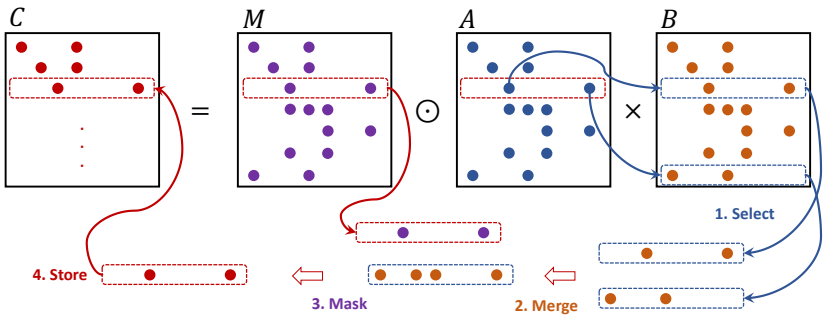


Fig. 2: The masked-SpGEMM $C = M \odot (A \times B)$, where $\odot$ is the element-wise multiplication and $\times$ is the usual matrix-matrix multiplication. The algorithm depicts the saxpy-based masked-SpGEMM, where each row of the C matrix is computed by first scaling the non-zero values of the corresponding row of A with the appropriate rows of B, and then filtering the intermediate result with the non-zero values from the corresponding row of M.

the computation. Lastly, we analyze the sparse accumulator used to store the intermediate results of the computation. Moreover, we extend the work by Milakovic et. al [17] by exploring the properties of the masking matrix and modifying the sparse accumulation. We believe that this study can offer important insights for library developers and sparse code generators alike, providing a recipe one needs to obtain efficient implementations for the masked-SpGEMM.

Contributions. We make the following contributions:

- We provide an in-depth analysis of the trade-offs for different iteration spaces, sparse accumulators, and tiling/thread scheduling schemes.
- We perform a performance study for a wide range of input graphs on a shared memory CPU system.
- We compare the performance against different setups and discuss our observations based on SuiteSparse:GraphBLAS and GrB.

The remainder of paper is structured as follows. Section II briefly describes the masked-SpGEMM kernel. Section III focuses on the three hypotheses that may influence performance. Section IV provides the experimental results and the discussion. Section VI we present related work. Finally, Section V summarizes the findings and outlines future directions.

II. Background

In this section, we briefly present the masked sparse-matrix-time-sparse-matrix (masked-SpGEMM) kernel. We then outline the current implementations both in the SuiteSparse:GraphBLAS library, but also in the GrB library, an implementation tailored for masked-SpGEMM.

A. Row-wise saxpy Masked SpGEMM

In this paper, we focus our analysis on the row-wise *saxpy*-based masked-SpGEMM algorithms, where all operands are stored in the CSR format.[1] By symmetry, our analysis also applies to column-wise *saxpy* over CSC operands. In the remainder of this paper, we refer to the row-wise *saxpy* simply as *saxpy*.

[1]We write *saxpy* in the BLAS $a\mathbf{x} + y$ sense.

```
# for each row of C
for i in 1 to m:
    # init accumulator
    acc = empty()
    # traverse all non-zero elements of A[i,:]
    for non-zero column k in A[i,:]:
        a = A[i,k]
        # fetch non-zero elements of B[k,:]
        for nonzero column j in B[k,:]:
            x = B[k,j]
            y = acc[i,j]
            acc[i,j] = a * x + y
    # intersect with mask
    for non-zero column j in acc[i,:]:
        if M[i,j] is zero:
            acc[i,j] = 0
    # store result to C
    C[i,:] = acc.gather()
```

Fig. 3: The *saxpy*-based masked-SpGEMM. The algorithm computes each row of the output matrix C in two steps. First, it scales the non-zero elements in the corresponding row of A with the appropriate rows of B. Second, the values of the results are element-wise multiplied with the non-zero elements of the row of M, via an intersection operation.

Let $A \in \mathbb{R}^{m \times K}$, $B \in \mathbb{R}^{K \times n}$, $M \in \mathbb{R}^{m \times n}$ be the input matrices stored using the CSR format. The masked-SpGEMM can be written as

$$C = M \odot (A \times B) \tag{1}$$

where $C \in \mathbb{R}^{m \times n}$ represents the sparse output matrix stored as well in the CSR format, $\odot$ represents the element-wise computation, and $\times$ represents the typical matrix-matrix multiplication. We use $\mathbb{R}$ here for simplicity, but GraphBLAS permits the use of any *semiring* instead.

Pictorially, the *saxpy*-based masked-SPGEMM is depicted in Figure 2, where A is a square matrix of size $m \times m$ and B and M are identical to A. Figure 3 outlines the pseudo-code for the vanilla masked-SpGEMM. The algorithm iterates over

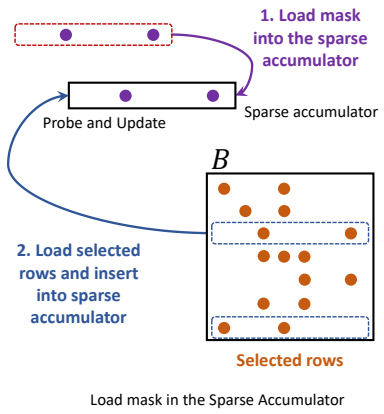


Fig. 4: The implementation of the masked-SpGEMM using a sparse accumulator. For each row of output matrix, the corresponding row from the mask is loaded into the accumulator. Each row loaded from the second matrix operand is intersected with the values stored in the accumulator. The non-zero elements of each row are filtered by the non-zero elements of the mask.

the m rows of output matrix C (line 1). For each row $C[i,:]$, the computation initializes a sparse accumulator as empty (line 4). For each non-zero element $A[i,k]$ of the corresponding row $A[i,:]$, the column index k is used to access the k-th row in matrix B (line 9). The non-zero values in $B[k,:]$ are scaled by $A[i,k]$ and added to the sparse accumulator (lines 10-12). Note that both the accumulator and $M[i,:]$ are sparse. Therefore, the output is obtained as an intersection between the two: masked elements are cleared from the accumulator. Finally, the result is packed and stored in C. In the following sections, we present the recent improved implementations and outline some key challenges in coming up with such implementations.

B. GraphBLAS and SuiteSparse:GraphBLAS

GraphBLAS is an API specification that defines the standard building blocks for graph algorithms in the language of linear algebra. For example, the definition

```
GrB_mxm(GrB_Matrix C,
        const GrB_Matrix M,
        const GrB_BinaryOp accum,
        const GrB_Semiring op,
        const GrB_Matrix A,
        const GrB_Matrix B,
        const GrB_Descriptor desc);
```

describes the function call and its arguments for computing a SpGEMM (`M` is set as `GrB_NULL`) or a masked-SpGEMM (`M` points to an actual `GrB_Matrix`). GraphBLAS works with opaque objects `GrB_Matrix` for the input `M`, `A`, `B` and output `C` matrices. Moreover, the function call requires the specifications of the semiring to outline the concrete operations. More details about the API can be found in [7], [8].

GraphBLAS specifies the API, SuiteSparse:GraphBLAS provides the highly optimized implementations for each API

```
1  # for each row of C
2  for i in 1 to m:
3      # init accumulator
4      acc = empty()
5      # load the mask into the accumulator
6      acc.setMask(M[i,:])
7      # traverse all non-zero elements of A[i,:]
8      for non-zero column k in A[i,:]:
9          a = A[i, k]
10         # fetch non-zero elements of B[k,:]
11         for nonzero column j in B[k,:]:
12             # accumulate if M[i,j] ≠ 0
13             if acc[i,j] is not masked:
14                 x = B[k,j]
15                 y = acc[i,j]
16                 acc[i,j] = a * x + y
17     # store result to C
18     C[i,:] = acc.gather()
```

Fig. 5: The modified masked-SpGEMM algorithm used by the GrB library. The algorithm first loads the mask in the accumulator. Then as each row from the `B` matrix is loaded, the non-zero values are checked in the accumulator to verify the mask is also non-zero. If there is a hit, the corresponding location is updated, if the mask has a zero element then the value is discarded.

function. SuiteSparse:GraphBLAS offers multiple implementations for the masked-SpGEMM computation. The library chooses between the different implementations using a handful of heuristics. While SuiteSparse:GraphBLAS attempts to offer a flexible solution to tackle different input graphs with different properties, the process of choosing between the implementations is automatic and hidden within the complexities of the library. Therefore, understanding how performance is obtained becomes a cumbersome task. In this paper, we attempt to offer insights that may shed some light on this.

C. GrB: an Optimized masked-SpGEMM Implementation

GrB [17] is a standalone library tailored for the masked-SpGEMM kernel, with a focus on the data structures used to store the intermediate results. GrB modifies the original implementation of the algorithm presented in Figure 3, by making the observation that before computing each row of `C`, the corresponding row of `M`, the mask, is loaded into the accumulator. Subsequent updates to the accumulator first verify that the non-zero values loaded from the second matrix hit within the mask as outlined in Figure 5. If the non-zero value has a corresponding non-zero in the loaded mask row, then the location is updated accordingly. If the non-zero value does not hit in the mask, then the value is discarded. In other words, this implementation intersects the each B row with the mask, and only updates the corresponding match in the accumulator as outlined in Figure 4. This approach is now used in SuiteSparse:GraphBLAS as well [18].

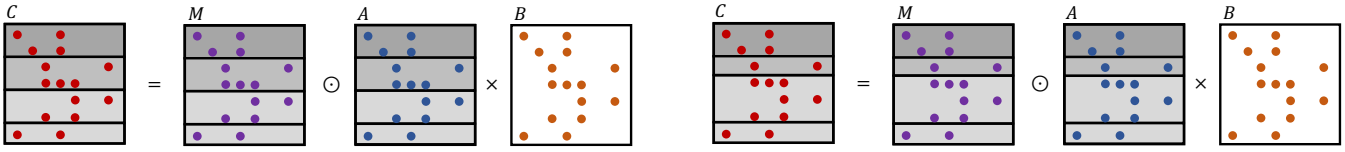


Fig. 6: The different tiling strategies for the *saxpy*-based masked-SpGEMM. Sub-figure (1), tiles the computation in the row dimension using homogeneous tiles. Each tile roughly has the same number of rows. Sub-figure (2), computes the total number of operations performed by the masked-SpGEMM. The tiles are then created based on the average number of operations. The goal of this approach is to load balance the computation.

GrB uses multiple data structures for storing the intermediate results. The most effective accumulators are either a Hash-based or Dense-based accumulators. For more details on their implementation, we recommend the reader to follow the paper [17]. The library offers some flexibility in choosing the different accumulators. However, the current implementation does not allow to choose different parallelization schemes. Given p threads, the implementation creates p tiles for the output C, the mask M and first operand matrices A. The second operand B is never tiled. The tiles are created by computing an average on the number of operations for each input matrix. The goal is to split the computation in tiles that balance the overall computation. The tiling and parallelization scheme is hence fixed.

III. Three Dimensions for Performance

In this section, we focus on the three main dimensions we identify as key in achieving efficient implementations for the masked-SpGEMM kernel. First, we outline techniques to tile the computation to achieve balanced execution across multiple threads. Second, we focus on the iteration space used to traverse the computation. Finally, we talk about the sparse accumulators used to store the intermediate results.

A. Tiling and Scheduling the Computation

As the regular SpGEMM is already a highly irregular and data-dependent computation, introducing masking further exacerbates the problem. When computing in parallel, achieving load balance between threads is critical to ensure effective hardware utilization. There are two main approaches to load balancing: (1) dynamic, and (2) static.

In the first case, a runtime system (e.g. OpenMP) schedules threads to remaining tasks as soon as they complete their current task. Load balance can be achieved in this case even when tasks are highly imbalanced, as long as there are sufficient independent tasks to assign to threads. However, the runtime system may incur additional overhead. In the second case, the tasks are scheduled offline and no runtime load balancing is used. This is common when tasks are balanced.

Ignoring the mask M for the moment, it is possible to compute the number of operations required to compute $C = A \times B$ in $O(nnz(A))$ time. Specifically, for each nonzero $A[i,k]$, we require $O(nnz(B[k,:]))$ operations. Since B is stored in CSR, $nnz(B[k,:])$ is available in constant time. Following the algorithm in Figure 5, we can estimate the work for a row as

$$W[i] = nnz(M[i,:]) + \sum_{A[i,k] \neq 0} nnz(B[k,:]). \qquad (2)$$

Using this, we can partition C into "FLOP-balanced" tiles. GrB uses this approach to create p tiles.

A simple alternative approach is to simply cut up C into uniformly sized tiles without regards to work, and let dynamic runtime scheduling do the balancing. It is also possible to combine both approaches, by producing $T > p$ balanced tiles and using dynamic scheduling. Based on our experience, SuiteSparse:GraphBLAS uses $T = 2p$ balanced tiles this way.

```
# for each row of C
for i in 1 to m:
    # init accumulator
    acc = empty()
    # traverse all non-zero elements of A[i,:]
    for non-zero column k in A[i,:]:
        a = A[i, k]
        # co-iterate M[i,:] with B[k,:]
        for nonzero column j in M[i,:]:
            # look up j in B[k,:]
            found = binary_search(B[k,:], j)
            if found:
                x = B[k, j]
                y = acc[i, j]
                acc[i,j] = a * x + y
    # store result to C
    C[i,:] = acc.gather()
```

Fig. 7: The masked-SpGEMM algorithm that co-iterates $M[i,:]$ with every row $B[k,:]$. Instead of looking the non-zero elements in $B[k,:]$, the algorithm looks at the non-zero elements in $M[i,:]$. It then uses a binary search to find the column index j in $B[k,:]$. If found it performs the computation.

B. Traversing the Computation

The masked-SpGEMM is never implemented as a two step operation, where the SpGEMM is done first, followed by the masking operation. Typically, the computation is performed in one step as outlined in Figure 3 and Figure 5. In the plain vanilla implementation, the masking operation is performed

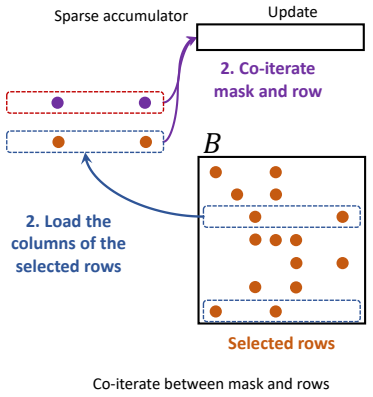


Fig. 8: The implementation of the masked-SpGEMM using the mask to co-iterate the rows of the A matrix. Only the common elements between the mask and each row are loaded and pushed into the accumulator. This approach is preferred, if the number of non-zeros in the mask is small compared to the number of non-zero elements in the loaded rows from A.

after all the rows $B[k,:]$ are merged and stored in the accumulator. The masking operation, represented by $M[i,:]$, filters out the values in the accumulator and outputs the final result to the C matrix. This approach requires a large buffer to store all the possible non-zero values obtained from merging the scaled $B[k,:]$ rows, and incurs many wasted computations.

The second implementation solves this problem by loading the mask $M[i,:]$ into the accumulator before computation is started. As the rows $B[k,:]$ are loaded from memory, the non-zero values from $B[k,:]$ are searched within the mask. If there is a hit, then the values in the accumulator are update accordingly, otherwise the non-zero value is discarded. The approach reads all the non-zero values from each $B[k,:]$, which may be problematic if $nnz(M[i,:])$ is significantly smaller than $nnz(B[k,:])$.

Using the mask $M[i,:]$ to co-iterate across the $B[k,:]$ rows as depicted in Figure 8, may reduce the amount of data loaded from main memory. The algorithm presented in Figure 7 outlines the approach. First, the mask is searched within the $B[k,:]$ row. If the mask is found within the row, then only the corresponding values are loaded from main memory. Unfortunately, this approach is not universally applicable. Co-iterating the mask $M[i,:]$ with the $B[k,:]$ rows works for masks with a small $nnz(M[i,:])$. If the $nnz(M[i,:])$ is high then the overhead of searching the mask every time within each $B[k,:]$ row may introduce a large overhead. More precisely, the cost to co-iterate $M[i,:]$ with $B[k,:]$ is given by

$$W_{co}[i,k] = nnz(M[i,:]) \cdot \log nnz(B[k,:]). \tag{3}$$

In order to intelligently switch between the two approaches, one can simply compare the cost $W_{co}[i,k]$ with $nnz(B[k,:])$ when fetching data from B. This motivates the algorithm shown in Figure 9. SuiteSparse:GraphBLAS internally uses this approach, and refers to it as a form of push-pull optimization [18]. Note that any co-iteration requires the B matrix columns to be sorted, which may not be the case in SuiteSparse:GraphBLAS.

```
# for each row of C
for i in 1 to m:
    # init accumulator
    acc = empty()
    # load the mask into the accumulator
    acc.setMask(M[i,:])
    # traverse all non-zero elements of A[i,:]
    for non-zero column k in A[i,:]:
        a = A[i, k]
        if W_co[i,k] < κ · nnz(B[k,:]):
            # co-iterate M[i,:] with B[k,:]
            for nonzero column j in M[i,:]:
                # look up j in B[k,:]
                found = binary_search(B[k,:], j)
                if found:
                    x = B[k, j]
                    y = acc[i, j]
                    acc[i,j] = a * x + y
        else:
            # fetch non-zero elements of B[k,:]
            for nonzero column j in B[k,:]:
                # accumulate if M[i,j] ≠ 0
                if acc[i,j] is not masked:
                    x = B[k,j]
                    y = acc[i,j]
                    acc[i,j] = a * x + y
    # store result to C
    C[i,:] = acc.gather()
```

Fig. 9: Hybrid linear scan and co-iteration. κ is the co-iteration factor, which trades off more or less co-iteration.

C. Sparse Accumulators

The sparse accumulator stores the partial sums during the computation of $C[i,:]$, and encodes the mask $M[i,:]$ to enable linear scanning of the B rows. The most important requirement of the accumulator is fast random access to *all possible output column indices*. There are two popular approaches to implement this: (1) a dense vector of size m, and (2) a sufficiently large hash table. Existing implementations, including GrB and SuiteSparse:GraphBLAS, use the operation count $\max_i \sum_{A[i,k]\neq 0} nnz(B[k,:])$, but we simply use $\max_i nnz(M[i,:])$ in our implementation. This is due to the fact that, with masking, we can have at most $\max_i nnz(M[i,:])$ output nonzeros. Furthermore, we need to set at least $\max_i nnz(M[i,:])$ elements in the beginning for the mask. Note that the max can be taken over the subset of rows owned by the thread, if using static scheduling. The dense accumulator may be preferred when the dimension of the matrix is small, or when there is significant spatial locality in the writes. On the other hand, the hash accumulator is often more space efficient when the dimensions are large, which can increase cache locality.

Name	Kind	n	nnz
arabic-2005	W	22,744,080	639,999,458
as-Skitter	W	1,696,415	22,190,596
circuit5M	C	5,558,326	59,524,291
com-LiveJournal	S	3,997,962	69,362,378
com-Orkut	S	3,072,441	234,370,166
europe_osm	R	50,912,018	108,109,320
GAP-road	R	23,947,347	57,708,624
hollywood-2009	S	1,139,905	113,891,327
stokes	C	11,449,533	349,321,980
uk-2002	W	18,520,486	298,113,762

TABLE I: Matrices used from the SuiteSparse Matrix Collection. The kinds are: (W) web graph, (C) circuit simulation, (S) social graph, (R) road graph.

A secondary requirement of the accumulator is fast state resetting between rows. In GrB, all $M[i,j] \neq 0$ slots of the accumulator are reset explicitly after each row. With SuiteSparse:GraphBLAS, a 64-bit marker is used for the dense accumulator to indicate which values are valid or invalid. After each row, the marker is incremented accordingly to implicitly reset the accumulator state. It is assumed that the marker does not overflow. Our modification of GrB uses the marker-based approach from SuiteSparse:GraphBLAS, except we relax the marker to be less than 64 bits. This may lead to overflow during marker increment, so overflow is detected and the state is fully reset when it occurs. This trades off the size of the state vector with the time taken to reset the vector. A smaller marker type can result in better locality of the state array, but also results in more time spent resetting the full array.

IV. Experimental Results

In this section, we describe our experimental setup. We briefly present the matrices used in our experiments. We then summarize the results for the three dimensions tat influence performance as described earlier.

A. Experimental Setup

Our evaluation focuses on masked-SpGEMM, where the input matrices M and B are identical to A. A is a sparse square matrix of size $m \times m$, where m represents the number of vertices in the input graph. For all the experiments, we fix the matrix A (the input graph) and compute $C = A \odot (A \times A)$, the main kernel used in triangle counting, k-truss. Following the GraphBLAS API, the mask is treated as Boolean (i.e., its values are not used).

All of our experiments are executed on a single node of the CPU partition on the NERSC Perlmutter supercomputer. Each node has 2 AMD EPYC 7763 CPUs, with 64 cores per socket partitioned into 4 NUMA domains (so 8 total), as well as 512GB of DRAM. In order to reduce NUMA effects, we restrict our experiments to run on a single socket, using 64 threads pinned to cores via OpenMP environment variables. Lastly, we use `numactl` to interleave all memory allocations across the 4 NUMA domains. We experimented with different configurations of allocating the memory using `numactl`, however all the experiments produced worse results compared to the interleaved case. As such, we do not report the results for the other configurations.

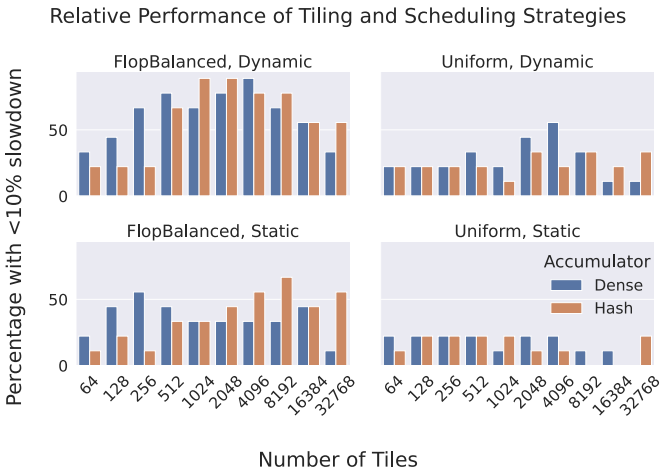


Fig. 10: Relative performance of different tiling and scheduling strategies, relative to the best. For each matrix, each configuration (split by accumulator) is compared to the lowest runtime for that matrix. The percentage corresponds how often each configuration was within 10% of the best configuration, across all matrices. Higher percentage is better.

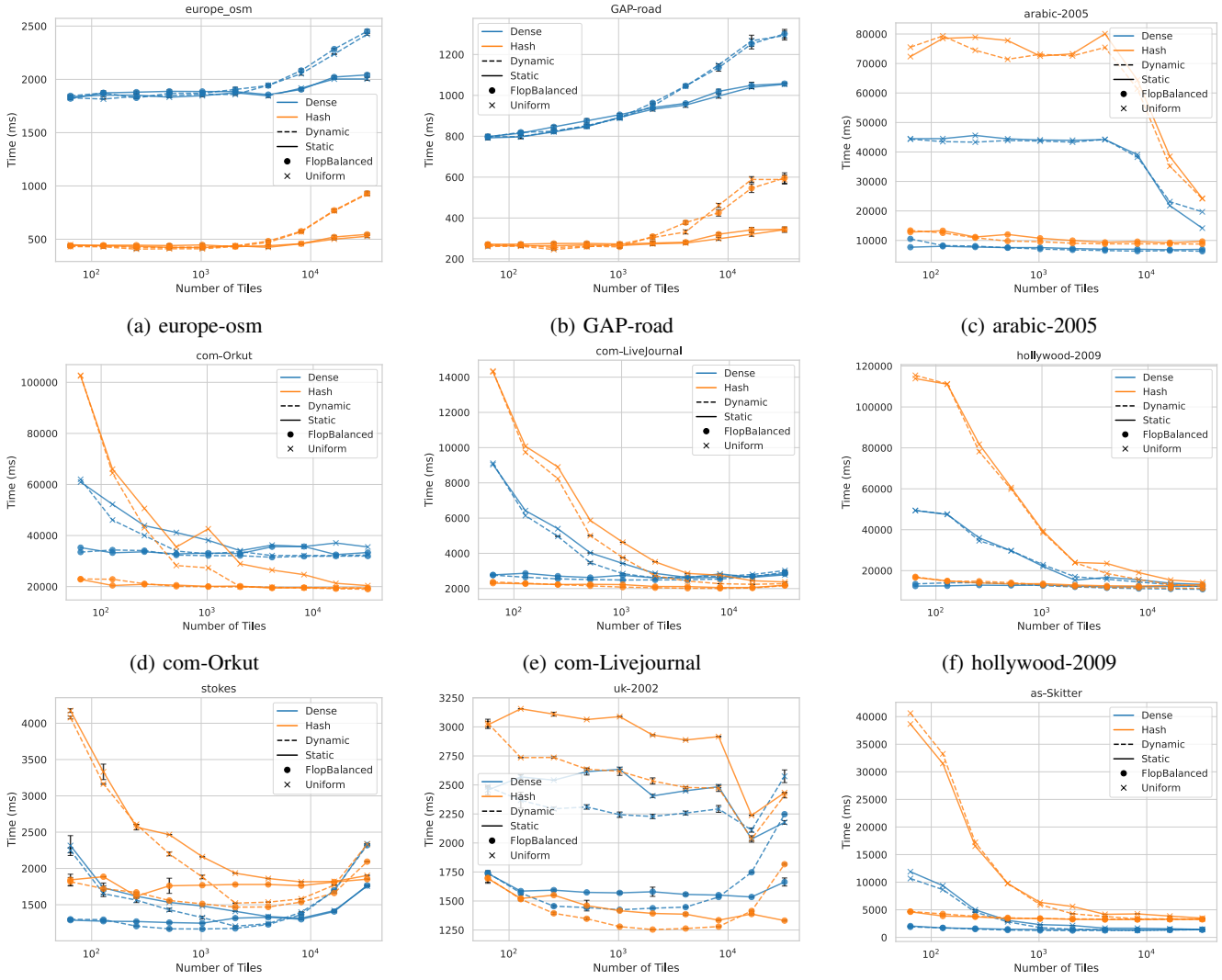
We have tested the masked-SpGEMM kernel with three different implementations. First, we compile GraphBLAS version 7.3 using the provided configuration file. Second, we use the GrB library to perform some of the benchmarks. Similar to the GraphBLAS variant, we compile the code using the predefined configuration file. Finally, we modify the GrB library and create our own version of the code. We parameterize this implementation to change the threads and tiles, and implement the dynamic approach that chooses between co-iterating the mask with the rows or loading the mask in the accumulator. For each experiment, we run the masked-SpGEMM kernel once for warm-up, then for 5 seconds or 10000 iterations, whichever comes first. The output is freed after each run.

B. Matrices

The matrices used in our experiments are summarized in Table I. We tried to select matrices with different characteristics. First, we looked at matrices from different domains. Our selection focuses primarily on various network graphs, including web hyperlink networks and social networks. We also include graphs from circuit simulation, as well as road graphs which are known to have unique performance characteristics. Second, we looked at large matrices that cannot easily fit within the last level cache of the AMD CPU (128 MB of L3 cache). Unlike many prior works including [17], we opt for relatively large matrices with tens to hundreds of millions of non-zeros. Performance in this regime is increasingly important as data and graph sizes continue to grow.

C. Tiling and Scheduling

We briefly present the results obtained when tiling the computation using the two techniques (flop-balanced vs ho-



(a) europe-osm (b) GAP-road (c) arabic-2005

(d) com-Orkut (e) com-Livejournal (f) hollywood-2009

(g) Stokes (h) uk-2002 (i) as-Skitter

Fig. 11: Results outlining the execution time in milliseconds for the masked-SpGEMM computation on the AMD EPYC CPU using 64 threads. The output, mask and first operand matrices are tiled. The second operand matrix is left as is. Each plot represents the execution time for one input graph. For each input graph, we use the hash and dense accumulators and the flop-balancing and homogeneous tiling. For each case, we increase the number of tiles from 64 to 32768. Moreover, we also report the different scheduling schemes when using OpenMP, namely static or dynamic. Lower execution time is better.

mogeneous tiling) outlined in Section III. For the masked-SpGEMM using both hash-based and dense accumulators, we report execution time in milliseconds. For these experiments, we did not include the co-iteration approach and focused on the algorithm presented in Figure 5. For each experiment, we perform a sweep on the number of tiles, ranging from 64 tiles to 32768 tiles. In addition, because we are using OpenMP, we experiment with `STATIC` and `DYNAMIC` scheduling of the tasks on the threads (each tile is assigned to one thread).

The results are shown in Figure 11. Note that some of the matrices exhibit similar behaviors. For example, the results for `europe-osm` and `GAP-road` are both road networks. As expected the experimental results outline that the two matrices exhibit the same trends. Similarly, `com-Orkut`, `com-Livejournal` and `hollywood-2009` are social network graphs and once again experience the similar behaviors. The other three matrices are outliers. `Stokes` is a circuit simulation, whole `arabic-2005` and `uk-2002` are directed graphs. For the `circuit5m` matrix we do no report tiling results, because the algorithm takes a significant amount of time and it times out on Perlmutter. In the following section, we will provide a discussion about our findings.

D. Hybrid Approach for Masking the Computation

For this set of experiments, we aim at investigating whether the adaptive algorithm described in Figure 9 provides better results compared to running the algorithm that does not co-iterate the mask. Similar to the previous set of experiments,

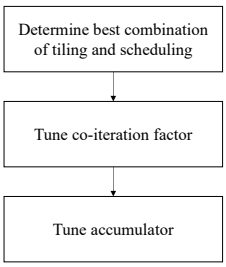


Fig. 12: Performance sweep and tuning flow. We sweep over the tiling and scheduling schemes without co-iteration. We determine the co-iteration factor κ and then tune the internal state representation of the accumulators.

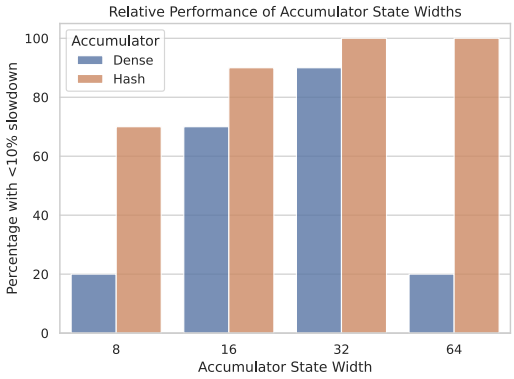


Fig. 13: Relative performance of different accumulator state bit-widths. The same methodology is used as in Figure 10.

we report execution time in milliseconds for both types of accumulators. We fix the number of tiles, the tiling strategy and the scheduling strategy. Based on the previous results we choose the configuration that provides the best execution time. We vary the co-iteration factor defined in Section III. We sweep across a range of co-iteration factors for both accumulators.

The results are shown in Figure 14. In this figure, we pick four representative matrices. For example, we choose `GAP-road` from the road network group of matrices. `europe-osm` exhibits the same behaviour. Note that in this set of experiments, we report execution time for the `circuit5M`. Recall that simply tiling the computation and using either of the accumulators, the algorithm in Figure 5 timed out. However, using the hybrid algorithm the execution time is reduced to 0.5 seconds using 64 threads and the co-iteration factor equal to 0.1. In the following section, we will provide more details about the results.

E. Accumulator State Tuning

For this experiment, we fix co-iteration factor $\kappa = 1$, and then we sweep over the marker size from 8 to 64 bits. The goal of this experiment is to try to reduce the size of the spare accumulators. The results are summarized as a relative performance plot in Figure 13.

V. Discussion

In this section, we discuss the key takeaways from the experimental results. Our experimental process follows the flow shown in Figure 12. We first sweep over the tiling and scheduling scheme without co-iteration to establish a safe choice for the remaining parameters. Then, we determine the ideal co-iteration factor κ. Finally, we tune the internal state representation for the accumulators, and discuss trade-offs between hash and dense.

A. Tiling the Computation

Based on Figure 11, we make the following observations:

1) Balanced tiling performs no worse than uniform tiling.
2) Uniform tiling performs poorly with lower tile counts, and can only match balanced tiling at higher tile counts.
3) Both tiling approaches can suffer at high tile counts.
4) Balanced tiling with an intermediate tile count and dynamic scheduling works generally well.

This suggests that the work calculation from Equation 2 is indeed a good estimate of load. However, there are occasionally imbalances that necessitate finer tiling, which is then exploited by dynamic scheduling. Figure 10 summarizes this nicely: depending on the accumulator, between 80-90% of matrices run with 2048 tiles, balanced tiling, and dynamic scheduling are within 10% of the best configuration.

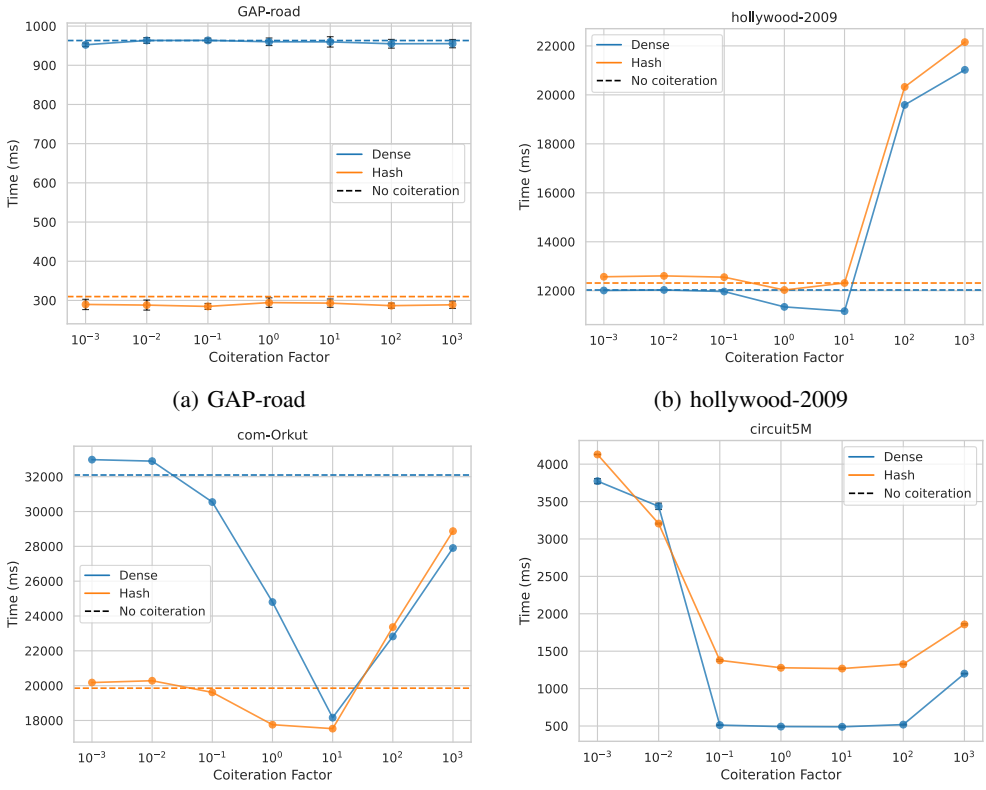
For these experiments, we only focused on tiling the computation in the row dimension. The matrices are stored as CSR, therefore no pre-processing steps are needed to perform expensive tiling operations. In addition, we did not perform any pre-processing of the data like partitioning the graphs, or reorganizing the data. For future work, we will investigate other data formats than CSR and possibly extend the experimentation to two dimensional tiling.

B. Iterating through the Data

Having fixed the tile count, tiling, and scheduling, we now turn to tuning the co-iteration factor κ. As seen in Figure 14, co-iteration has a minimal effect on the GAP-road network, while both positive and negative effects are present away from $\kappa \approx 1$ in the other networks. The `circuit5M` matrix is of particular interest, as the baseline without co-iteration did not complete within a reasonable time. The `com-Orkut` matrix exhibits a nearly $2\times$ reduction in runtime with the dense accumulator, matching the hash accumulator. This is likely due to less cache evictions caused by reading large chunks of the dense accumulator and B rows. Generally, the results indicate that the estimate from Equation 3 is accurate relative to the linear estimate from Equation 2, and that no significant scaling factor is needed.

C. Accumulator

Finally, we tune the marker bit-width. As Figure 13 shows, the hash accumulator is somewhat robust to the bit-width,



(a) GAP-road

(b) hollywood-2009

(c) com-Orkut

(d) circuit5M

Fig. 14: Results outlining the execution time in milliseconds for the masked-SpGEMM computation with varying the co-iteration factor. We fix the number of tiles to 2048 tiles, we fix the OpenMP scheduling policy to DYNAMIC, and we use the FLOP-balanced tiling scheme. We show results for four representative matrices. The thick lines represent the algorithm that utilized the co-iteration, while the dotted lines represents the execution of the non co-iterate algorithm. For this set of experiments we report the execution time for `circuit5M` which timed out for the non co-iteration algorithm.

maintaining similar performance until 8 bits where it degrades. On the other hand, the dense accumulator suffers at both 8 and 64 bits, with a sweet spot at 32 bits. This reaffirms the importance of tuning for memory efficiency.

VI. Related Work

A. GraphBLAS API and Implementations

The GraphBLAS project [7] aims to provide a set of linear-algebraic (i.e. matrix, vector, and scalar) primitives for expressing graph analytics workloads, in the spirit of the dense BLAS specification. SuiteSparse:GraphBLAS provides the canonical implementation of the GraphBLAS API [8], [18]. GraphBLAST provides a performant implementation of GraphBLAS on GPUs [19].

B. Masked-SpGEMM

Azad et al. [20] was the first to define the masked-SpGEMM primitive operation in the context of linear algebraic graph analytics. As with our analysis, they focused on triangle counting as the workload. Milaković et al. [17] explore a large space of sparse accumulators and higher-level algorithms beyond row-wise *saxpy*. We use their codebase as our starting point for exploring additional tiling strategies and the effect of co-iteration.

C. Database Query Planning and Execution

The masked-SpGEMM kernel can be viewed as a specific variant of the well-known *triangle query* in the databases community. More precisely, if matrices are viewed as binary relations, then the masked-SpGEMM kernel is equivalent to

$$C(i,j) = \prod_{i,j} M(i,j) \bowtie A(i,k) \bowtie B(k,j),$$

where joins and projections are interpreted in the framework of K-relations over semirings [21].

The triangle query is notorious for being inefficient to compute using binary joins, taking $O(m^2)$ time when the result is of size $O(m^{3/2})$ where $m = nnz(A)$. Analogously, post hoc masking an unmasked-SpGEMM (which is really a binary join) suffers from the same issue. Under this lens, *efficient* implementations of masked-SpGEMM are actually instances of *worst-case optimal join* [22]. In fact, the dynamic co-iteration strategy described here is exactly the mechanism Generic Join [23] uses to achieve worst-case optimality: iterate over the smaller of two relations for every intersection computed (ignoring logarithmic access costs).

VII. Conclusion

In this work, we focused on providing insights into how to obtain performance for the masked-SpGEMM, a widely used sparse linear algebra kernel. We started with three dimensions, namely 1) tiling the computation and distributing the tiles across threads, 2) deciding on how to iterate through the data to reduce the amount of data moved from memory and 3) choosing the right accumulator to store the intermediate results. Based on the experimental results, we can state that for the *saxpy*-based implementation tiling the computation using a good load balancing estimator and using the DYNAMIC policy for the OpenMP scheduling are important. In addition, co-iterating across the mask may provide significant improvements for certain input graphs. Finally, designing and tuning the accumulator can reduce the temporary buffers and improve execution time.

Stepping back, we have taken a staged approach to analyzing and tuning the three performance dimensions. These dimensions are inevitably correlated, and likely in complex ways. With the data we have gathered, we intend to perform a more precise analysis of the effects matrix structure and features have on the different parameters. Ideally, this data will enable us to build models which can intelligently tune the parameters at execution time, rather than offline for the average case.

VIII. Acknowledgements

We thank Srđan Milaković for his helpful discussions on masked-SpGEMM performance, as well as assistance in setting up and working with the GrB codebase. We also thank the reviewers for their time and their valuable feedback that helped us improve the paper. This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the X-STACK: Programming Environments for Scientific Computing program (DE-SC0021982), along by the National Science Foundation through grants IIS-1955488, IIS-2027575, DOE award DE-SC0016260, ARO award W911NF2110339, and ONR award N00014-21-1-2724. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

References

[1] A. Kyrola, G. Blelloch, and C. Guestrin, "{GraphChi}:{Large-Scale} graph computation on just a {PC}," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 31–46.

[2] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[3] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, 2018, pp. 752–768.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[5] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[6] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[7] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[8] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[9] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 804–811.

[10] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[11] T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan, "First look: Linear algebra-based triangle counting without matrix multiplication," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–6.

[12] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan *et al.*, "Quickly finding a truss in a haystack," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[13] O. A. Rodriguez, Z. Du, J. Patchett, F. Li, and D. A. Bader, "Arachne: An arkouda package for large-scale graph analytics," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[14] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan, "Linear algebraic formulation of edge-centric k-truss algorithms with adjacency matrices," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[15] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–10.

[16] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[17] S. Milaković, O. Selvitopi, I. Nisa, Z. Budimlić, and A. Buluç, "Parallel algorithms for masked sparse matrix-matrix products," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.

[18] T. A. Davis, "Algorithm1037: Suitesparse:graphblas: Parallel graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. 49, no. 3, sep 2023. [Online]. Available: https://doi.org/10.1145/3577195

[19] C. Yang, A. Buluc, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," 2021.

[20] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 804–811.

[21] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 31–40. [Online]. Available: https://doi.org/10.1145/1265530.1265535

[22] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," 2012.

[23] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: new developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, p. 5–16, feb 2014. [Online]. Available: https://doi.org/10.1145/2590989.2590991