École polytechnique fédérale de Lausanne

Master thesis spring 2023

Master in Computational science and Engineering

Efficient graph algorithms

Author:
Bruno Rodriguez Carrillo

Supervisor:
Daniel Kressner

Contents

Chapter 1

Introduction

GraphBLAS is a framework for graph algorithms that are expressed as linear algebra operations. It provides a high-level interface for performing graph algorithms and computations on sparse graphs using sparse matrix operations.

The key idea behind GraphBLAS is to leverage the mathematical foundations of linear algebra to provide a unified framework for graph computations. By representing graphs as sparse matrices, graph algorithms can be expressed as matrix operations, allowing for efficient and scalable implementation on parallel architectures.

GraphBLAS supports a wide range of graph algorithms like breadth first search or Prim's algorithm. Following the BLAS standard, GraphBLAS classifies operations on scalars, vectors and matrices as level-1, level-2 and level-3 operations, respectively.

There exist several implementations GraphBLAS, many of which follow the GraphBLAS API C standard as described in [2]. In GraphBLAS, there are different functions to represent operations on matrices, vectors or scalars. Algorithms written in GraphBLAS correspond to a sequence of function calls, the tasks of which must be performed following the order into which they appear in the program.

The GraphBLAS API C establishes two execution mode for each GraphBLAS function. In the *blocking* mode, invoking each of the functions implies its execution; that is, once a function is called all computation the function is responsible for are performed and results are written into memory and then the function returns. In this mode, calling a function implies its fully execution. Moreover, each function is executed in the order it is found in the program.

On the other hand, the *nonblocking* mode, calling a function does not necessarily imply its execution. That is, a function in this mode can return without having performed its operations. The main idea of the nonblocking mode is to delay the actual execution of GraphBLAS functions so that they are executed whenever it is necessary for correct results in the program.

ALP/GraphBLAS is an implementation that follows the GraphBLAS API C which is written in `C++11`. In [8], the nonblocking mode is implemented for level-2 operations in ALP/GraphBLAS. In [8], the authors present how the nonblocking mode can be enabled primarily by using a data dependence analysis that checks what data each of the GraphBLAS functions in the program accesses to so that it is possible to determine whether execute or delay the execution of the functions.

However, in the current nonblocking mode implemented in ALP/GraphBLAS, level-3 operations are not considered. As a result, in this work, by following the ideas presented in [8], the current design and implementation of the nonblocking mode in ALP/GraphBLAS are extended such that operations that output matrices are supported; special emphasis is found on the sparse matrix-sparse matrix multiplication.

This work is organized as follows. Chapter 2 and 3 present the required mathematical background that supports GraphBLAS and the most important ideas of what GraphBLAS is, respectively. Chapter 4 explains in detail the key components of the nonblocking execution mode for level-2 operations presented in [8].

In Chapter 5, it is explained how to extend the current implementation of the sparse matrix-sparse matrix product algorithm in ALP/GraphBLAS to be executed in the nonblocking mode. Chapter 6 presents the performance of a nonblocking-executed sparse matrix-sparse matrix multiplication algorithm when applied to the triangle counting problem. Finally, Chapter 7 displays future ideas to further develop the nonblocking mode in ALP/GraphBLAS.

Chapter 2

Mathematical background

2.1 Definitions

The following three definitions are taken from [7].

Definition 2.1.1 (Undirected graph)**.** An **undirected graph** is a triple (V, E, Ψ), where V and E are finite sets and $\Psi : E \to \{X \subseteq V : |X| = 2\}$. A **directed graph** or **digraph** is a triple (V, E, Ψ), where V and E are finite sets and $\Psi : E \to \{(v, w) \in V \times V : v \neq w\}$. The elements of V are called the **vertices** and the elements of E are the **edges**.

Two edges e_1, e_2 with $\Psi(e_1) = \Psi(e_2)$ are called **parallel**. Graphs without parallel edges are called **simple**. For simple graphs, an edge e is identified with its image $\Psi(e)$ and write $G = (V(G), E(G))$, where $E(G) \subseteq V(G) \times V(G)$. A simple graph is also called a graph with no self-loops.

When an edge e **joins** vertices v and w, we write $e = \{v, w\}$ or $e = (v, w)$; v and w are the endpoints of e. The vertices v and w are **neighbours** of each other. Then, v and w are called **adjacent**. For an undirected graph, if v is and endpoint of e, then v is **incident** with e. In directed graphs, it is said that (v, w) **leaves** v and **enters** w. If two edges share at least one endpoint, these are called **adjacent**.

Definition 2.1.2 (Incidence matrix)**.** The **incidence matrix** of an undirected graph G is the matrix $\mathbf{A} = \big(a_{v,e}\big)_{v \in V(G), e \in E(G)}$ where

$$a_{v,e} = \begin{cases} 1 & \text{if } v \in e, \\ 0 & \text{if } v \notin e. \end{cases}$$

Definition 2.1.3 (Adjacency matrix)**.** The **adjacency matrix** of an undirected graph G is the matrix $\mathbf{A} = \big(a_{v,w}\big)_{v,w \in V(G)}$ where

$$a_{v,w} = \begin{cases} 1 & \text{if and only if } \{v, w\} \in E(G), \\ 0 & \text{if } v \notin e. \end{cases}$$

2.2 Triangle counting

From this point forward, we assume that we are given the adjacency matrix $\mathbf{A}$ of an undirected or directed simple graph $G = \big(V(G), E(G)\big)$ that has n vertices. Simple graph means that there are no self-loops in G. That is, $\mathbf{A}(i, j) = 0$ for $i = j$.

We are applying the implementation presented in this work to the triangle counting problem. This problem consists of, given an undirected or directed simple graph $G = \big(V(G), E(G)\big)$ with n vertices, counting the total number of triangles for each node in the G.

We provide two useful theorems for the purposes of this work. No proofs are provided for either. We first state Theorem 2.2.1 that as found in [3] as Theorem 0.1.

Theorem 2.2.1. *The (i, j) entry of a_{ij}^k of $\mathbf{A}^k$, where $\mathbf{A}$, the adjacency matrix of G, counts the number of walks of length k having start and end vertices i and j*

Moreover, since we are assuming the G has no self-loops, it can be proven that the only paths of length three that can be found on G correspond to triangles. Then, we state Theorem 2.2.2 that as found in [9] as Theorem 1.1.

Theorem 2.2.2. *If G is a simple graph with adjacency matrix $\mathbf{A}$, then the number of 3-cycles in G is*

$$\textit{\#3-cycles in } G = \frac{1}{6}\sum_{i=1}^{n} tr\left(\mathbf{A}^3\right) \tag{2.1}$$

In Theorem (2.2.2), the number of 3-cycles in G refers to the number of path of length three in G.

We now provide an example to explain the ideas behind Theorem 2.2.1 and Theorem 2.2.2. Consider the 4-node simple, undirected graph G shown in Figure 2.1, its adjacency matrix $\mathbf{A}$ is given in (2.2), and $\mathbf{A}^3$ is given in (2.3).

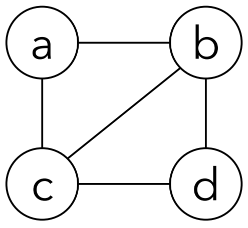


Figure 2.1: Undirected graph G.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \tag{2.2}$$

$$\mathbf{A}^3 = \begin{pmatrix} 2 & 5 & 5 & 2 \\ 5 & 4 & 5 & 5 \\ 5 & 5 & 4 & 5 \\ 2 & 5 & 5 & 2 \end{pmatrix} \tag{2.3}$$

From Figure 2.1, we observe that there are two triangles in G. Let us now explain how to get to this results from the previous theorems.

On the diagonal of matrix (2.3), we find the number of paths of length 3 or triangles that start and end at each of the vertices of the graph. For example, $\mathbf{A}^3(1, 1) = 2$ means that there are two triangles that start and end at vertex a, whereas $\mathbf{A}^3(2, 2) = 4$ shows that there are four triangles triangles that start and end at vertex b, and so on. This is the idea of Theorem 2.2.1.

Then, by computing the $\operatorname{tr}\left(\mathbf{A}^3\right)$ would lead to the total number of triangles in G. However, there is a single triangle that contains a as a vertex and two triangles that have b as a vertex.

The meaning of diving $\operatorname{tr}\left(\mathbf{A}^3\right)$ by 6, Theorem 2.2.2 is as follows. From 2.1, we observe that there is a first triangle that that starts and ends at vertex a, but $\mathbf{A}^3(1,1) = 2$. The reason for which this happens is that there are two paths of length 3 that start and end at vertex a: one that follows the path $a - c - b - a$, and the second one that follows $a - b - c - a$. Moreover, this means that triangle of vertices a, b and c is counted twice. We can use a similar explanation for $\mathbf{A}^3(4,4) = 2$.

The reason for which $\mathbf{A}^3(2,2) = 4$ is that vertex b belongs to two triangles and for each these triangles, there are two paths of length 2 that start and end at b. For the triangle of vertices a, b and c, one path corresponds to $b - a - c - b$ and the second one is $b - c - a - b$. This means that the triangle with vertices a, b and c is counted two more times in this case. On the other hand, for the triangle of vertices b, c and d, one path corresponds to $b - d - c - b$ and the second one is $b - c - d - b$. We could use the same reasoning to explain why $\mathbf{A}^3(3,3) = 4$.

Therefore, we observe that the triangle of vertices a, b and c is counted six times in total. Twice in $\mathbf{A}^3(1,1)$, twice $\mathbf{A}^3(2,2)$, and twice in $\mathbf{A}^3(3,3)$. We could then say that each triangle contributes 6 times total number of triangles in Figure 2.1. Similar ideas follow for the triangle with vertices b, c and d. This is an idea to explain why the sum in equation 2.1 is divided by 6.

Chapter 3

GraphBLAS

All definitions of GraphBLAS for vectors, matrices, operators, monoids and semirings below are explicitly taken from [2].

GraphBLAS is a framework that allows expressing graph algorithms as linear algebra operations [6].

One of the core concepts or ideas behind GraphBLAS is that it promulgates the usage of a common interface to express graph algorithms through operations involving scalars, vectors and matrices. The GraphBLAS library defines a set of algebraic operations, such as matrix multiplication, element-wise addition, and reduction, which can be seen as graph algorithms. It provides a programming framework that allows users to express graph algorithms in a precise and shorter manner.

3.1 The C GraphBLAS API

Herein, we present important concepts and definitions related to how the GraphBLAS API C is constructed and how it works. The GraphBLAS API C manipulates and works on objects represented as *opaque data types*, which means that the objects are manipulated by the GraphBLAS implementation only. From [2], these objects are:

- *Collections:* vectors and matrices
- *Algebraic objects:* unary and binary operators, monoids and semirings
- *Control objects:* descriptors and masks

Functions that manipulate GraphBLAS objects are called *methods.*

Definition 3.1.1 (GraphBLAS vector)**.** A GraphBLAS vector $\mathbf{v}$ is defined by three elements

$$\mathbf{v} = \langle D, N, \{(i, v_i)\}\rangle,$$

where D is called the domain of $\mathbf{v}$ and it corresponds to the data type of the elements stored in $\mathbf{v}$; the size or length of $\mathbf{v}$ is N; a set of tuples (i, v_i) where $0 \leq i < N$ and $v_i \in D$. We can think of the values i as the indices of $\mathbf{v}$, and each value of i must be appeared once only for any GraphBLAS vector -each index value i is unique for $\mathbf{v}$.

Definition 3.1.2 (content of a GraphBLAS vector)**.** The number of elements of $\mathbf{v}$ is defined as $\mathbf{nelem}(\mathbf{v}) = N$, and the content of $\mathbf{v}$ is defined as $\mathbf{L}(\mathbf{v}) = \{(i, v_i)\}$. For $\mathbf{v}$, $\mathbf{v}(i)$ corresponds to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$; otherwise, $\mathbf{v}(i)$ is undefined.

Definition 3.1.3 (GraphBLAS matrix)**.** A GraphBLAS matrix $\mathbf{A}$ is defined by four elements

$$\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle,$$

where D is called the domain of $\mathbf{v}$ and it corresponds to the data type of the elements stored in $\mathbf{A}$; the number of rows of $\mathbf{A}$ is M; the number of columns of $\mathbf{A}$ is N; a set of tuples $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$ where $0 \leq i < M$, $0 \leq j < N$ and $A_{ij} \in D$.

Definition 3.1.4 (GraphBLAS matrix)**.** The set $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$ is called the content of $\mathbf{A}$, and the number of rows and columns are $\mathbf{nrows}(\mathbf{A}) = M$ and $\mathbf{ncols}(\mathbf{A}) = M$, respectively. For $\mathbf{A}$, $\mathbf{A}(i, i)$ corresponds to A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$; otherwise, $\mathbf{A}(i, j)$ is undefined.

In GraphBLAS vectors and matrices, values that are not explicitly stored in these collections are undefined. This is important because by doing this, the confussion of interpreting the meaning of zero is avoided when different semirings are used. That is, having an undefined element in a container does not necessarily imply that its value is zero.

Definition 3.1.5 (GraphBLAS matrix)**.** The jth column of a GraphBLAS matrix $\mathbf{A}$ is the vector given by

$$\mathbf{A}(:, j) = \langle D, M, N, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle,$$

where $0 \leq j < N$. Similarly, the ith row of $\mathbf{A}$ is the vector

$$\mathbf{A}(i, :) = \langle D, M, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle,$$

where $0 \leq i < M$.

Definition 3.1.6 (transpose GraphBLAS matrix)**.** Given a GraphBLAS matrix $\mathbf{A}$, its transpose $\mathbf{A}^T$ is defined by

$$\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle.$$

Definition 3.1.7 (GraphBLAS binary operator)**.** A GraphBLAS binary operator is defined as

$$F_b = \langle D_1, D_2, D_3\rangle.$$

It has three domains D_1, D_2, D_3, and an operation $\odot$, which is defined as $\odot : D_1 \times D_2 \to D_3$.

Definition 3.1.8 (GraphBLAS unary operator)**.** A GraphBLAS unary operator is defined as

$$F_u = \langle D_1, D_2, f\rangle.$$

It has three domains D_1 and D_2, and an operation f, which is defined as $f : D_1 \to D_2$.

Definition 3.1.9 (GraphBLAS moniod)**.** A GraphBLAS moniod is defined as

$$M = \langle D_1, \odot, \mathbf{0}\rangle.$$

It has a single domain D_1, an associative operation $\odot : D_1 \times D_1 \to D_1$, and an identity element represented by $\mathbf{0}$. If $F_b = \langle D_1, D_1, D_1\rangle$ is a GraphBLAS binary operator, and $\mathbf{0}$ the identity for $\odot$, then

$$M = \langle F, \mathbf{0}\rangle = M = \langle D_1, \odot, \mathbf{0}\rangle$$

is the associated GraphBLAS monoid.

Definition 3.1.10 (GraphBLAS semiring)**.** A GraphBLAS semiring

$$S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0} \rangle$$

is defined by three domains D_1, D_2, and D_3, an associative additive operation $\oplus : D_3 \times D_3 \to D_3$, an multiplicative operation $\otimes : D_1 \times D_2 \to D_3$, and the identity element for $\oplus$, $\mathbf{0} \in D_3$.

Let us consider the GraphBLAS binary operator given by $F = \langle D_1, D_2, D_3, \otimes \rangle$ and the GraphBLAS monoid $M = \langle D_3, \oplus, \mathbf{0} \rangle$, thus $S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0} \rangle$ is a GraphBLAS semiring.

Similarly, for a given GraphBLAS semiring $S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0} \rangle$, there always exist its associated monoid $M = \langle D_3, \oplus, \mathbf{0} \rangle$, and its associated binary operator $F = \langle D_1, D_2, D_3, \otimes \rangle$.

It is important to notice that the multiplicative operation in the semiring definition within GraphBLAS allows inputs from two different domains that can produce outputs in a third distinct domain, and does not require defining the identity element for the multiplicative operation.

For the purpose of this work, we mention that the GraphBLAS API C establishes two execution mode for each GraphBLAS function. In the *blocking* mode, invoking each of the functions implies its immediate execution; that is, once a function is called all computation the function is responsible for are performed and results are written into the corresponding collections and then the function returns. In this mode, calling a function implies its fully execution. Moreover, each function is executed in the order it is found in the program.

On the other hand, the *nonblocking* mode, calling a function does not necessarily imply its execution. A function in this mode can return without having performed its operations. The main idea of the nonblocking mode is to delay the actual execution of GraphBLAS functions so that they are executed whenever it is absolutely necessary for correct execution of a program.

3.2 SuiteSparse:GraphBLAS

SuiteSparse:GraphBLAS is a `C` implementation that includes all the primitives described in the GraphBLAS API C, [2]. This is the reason for which it is called a full or complete implementation of the GraphBLAS API C. This implementation can be found in `https://github.com/DrTimothyAldenDavis/GraphBLAS`

3.3 ALP/GraphBLAS

The Algebraic Programming (ALP) project is a `C++11` programming framework that includes the following interfaces:

1. Generalised sparse linear algebra, ALP/GraphBLAS
2. Vertex-centric programming, ALP/Pregel

In this work, we focus on the first interface exclusively. ALP/GraphBLAS is a C++11 implementation of GraphBLAS that takes the GraphBLAS API C presented in [2] as a starting point and unifies it with `C++11` programming. There are two important characteristics of ALP/-GraphBLAS. Firstly, the ALP/GraphBLAS library is implemented in distinct template-based backends. This allows for the users to use the same API while having different code optimizations, which are enabled by each backend. Having different backends while keeping the same interface is a powerful idea that permits the user to write a single program that can be run with different code optimizations.

This is significant because the user is only responsible for writing the program; all implementation details are hidden to the user. Moreover, the idea behind this is that all code optimizations are automatically selected at compilation time depending on the backend of choice. What backend to use is decided at compilation time by the user; that is, once a program is written in ALP/GraphBLAS, the user is responsible for choosing with which backend to execute the program. Users are never required to provide a backend template argument explicitly to the functions used in a program.

In the current implementation of ALP/GraphBLAS and for the purposes of this work, we mention the three main backends: the serial or sequential-execution backend that vectorizes operations, the parallel backend that uses OpenMP, and the nonblocking execution backend. These backends lead to having different performances for a single program.

The reference backend corresponds to the blocking mode and the nonblocking backend implements the nonblocking mode as described in the GraphBLAS API C. All the functions implemented in the reference backend execute all operations they are responsible for before returning. This backend executes all its functions sequentially; no parallelization is performed; the reference backend corresponds to single-threaded, serial execution.

Furthermore, since the ALP/GraphBLAS API is the same for all the backends, each function in ALP/GraphBLAS is implemented for every single backend. The implementation details of each function changes when using a different backend but its usage by the user, i.e. how it is called in a program, remains the same no matter the backend of choice.

All ALP/GraphBLAS functions are preceded by the `grb` namespace name. Another important concept in ALP/GraphBLAS is the type elements each function manipulates. These can be scalars, vectors or matrices. Vectors are called 1D containers and matrices 2D containers. Matrices and vectors are exposed to the user as *opaque data types*, which means that they can accessed and modified by the ALP/GraphBLAS API exclusively. We find two opaque containers: vectors (written as `grb::Vector⟨T⟩`) and matrices (written as `grb::Matrix⟨T⟩`). Vectors can be sparse or dense, whereas matrix are always assumed to be sparse. Both containers are of a static type `T`, which specifies the data type of the elements the container stores, which must be any plain-old-data (POD) type. Following the GraphBLAS API C, scalars are not implemented as an opaque data type in the current version of ALP/GraphBLAS. In this work, we usually refer to vector and matrices simply as containers and vice-versa when the context is clear, and we call primitives all ALP/GraphBLAS functions that access to or modify containers.

Depending of the dimensions of the container to which a primitive accesses, in ALP/GraphBLAS, we distinguish four type of primitives.

- Level-0 Primitives. A set of ALP/GraphBLAS primitives that can work on zero-dimensional containers (scalars)

- Level-1 Primitives. A set of ALP/GraphBLAS primitives that can handle a mix of zero-dimensional and one-dimensional containers (`grb::Vector`), e.g., vector-vector operations

- Level-2 Primitives. A set of ALP/GraphBLAS primitives that handle operations between zero-dimensional, one dimensional, and two-dimensional containers (`grb::Matrix`), e.g., matrix-vector operations

- Level-3 Primitives. A set of ALP/GraphBLAS primitives that allow working with operations between two-dimensional containers, e.g., matrix-matrix operations

Additionally, another important concept is the *capacity* of a container. This is defined as the total number of nonzero elements a container can hold at most. Closely related to this concept is the *number of nonzeros* in a container, which is defined as the actual number of nonzeros that a container stores. The number of nonzeros must be always smaller or equal to the capacity of the container. In case of a vector, if its number of nonzeros is smaller than its capacity, it is considered as sparse; it is dense otherwise. Scalars correspond to the standard `C++11` plain-old-data-types, and consequently, scalars have size and capacity equal to one, and they are always dense.

In ALP/GraphBLAS, the user can find a collection of standard operators and identities that satisfy the algebraic properties given in section 3.1, which can be fused together to define a large variety of monoids and semirings.

In addition, in ALP/GraphBLASall operators and data types of the containers are checked using `C++11` type traits at compilation time, which allows error handling at compilation time rather than at run time.

There are primitives presented in the GraphBLAS API C that are not implemented in ALP/GraphBLAS and level-3 operations are not fully implemented to be executed in parallel. The current implementation of ALP/GraphBLAS is found on `https://github.com/Algebraic-Programming/ALP`.

Chapter 4

Nonblocking execution mode

4.1 Notes on notation

From this chapter and forward, we represent matrices with bold upper cases, i.e, $\mathbf{A}$, vectors with bold lower cases, i.e $\mathbf{v}$, and scalars lower cases, i.e x.

All matrices are assumed to be of the form $\mathbf{A} \in \mathbb{R}^{n \times n}$. We use $nnz\left(\mathbf{A}\right)$ to denote the number of nonzero elements elements in $\mathbf{A}$. Similarly, all vectors are assumed to be of the form $\mathbf{v} \in \mathbb{R}^{n}$; scalars are always of the form $x \in \mathbb{R}$. All operations with matrices, vectors and scalars correspond to standard arithmetic.

We use the 0-based indexing for matrices and vectors throughout this work. For sparse matrix indexing, we use the MATLAB colon notation; $\mathbf{A}\left(:, i\right)$ denotes the i-th column of $\mathbf{A}$, $\mathbf{A}\left(i, :\right)$ denotes the i-th row of $\mathbf{A}$, and $\mathbf{A}\left(i, j\right)$ denotes the (i, j)-th element of $\mathbf{A}$; concerning vectors, $\mathbf{v}\left(i\right)$ denotes the i-th element of $\mathbf{v}$.

To denote matrices, vectors, and scalars that are stored on a computer, we do not employ any different notation. For example, we will use the notation $\mathbf{A}$ to refer to a matrix stored on a computer's memory. Lines of code or operations that happen in a program are written using this font: `example of line of code`.

Moreover, for each mathematical operation that involves matrices, vectors or scalars, there is a GraphBLAS primitive that performs that operation. Then, we may use mathematical notation to refer to a GraphBLAS primitive when the context is clear. For instance, the notation $\mathbf{A} = \mathbf{BC}$ may refer to the GraphBLAS primitive responsible for computing that product.

We denote the number of nonzeros of a matrix $\mathbf{A}$ and its capacity (maximum number of nonzeros a matrix can hold) as $nnz(\mathbf{A})$ and $cap(\mathbf{A})$, respectively. The same notation applies to vectors.

4.2 Essential components of the nonblocking mode for level-2 primitives

In [8], it is presented the design and implementation of the nonblocking backend in ALP/GraphBLAS for shared-memory systems. This implementation handles level-1 and level-2 primitives in nonblocking mode.

In this chapter, the main ideas presented [8] to achieve a nonblocking backend are analyzed so that similar ideas can be extended and applied to level-3 primitives, which is detailed in next chapter. There are four key components that enable having nonblocking execution: lazy evaluation, loop fusion, loop tiling, and loop parallelization.

At the end of this chapter, it is explained how the primitives in a program are executed in the nonblocking backend by presenting and discussing the Conjugate Gradient algorithm implemented in ALP/GraphBLAS.

Loop fusion is a compiler optimization and loop transformation that replaces multiple loops in a program with a single one. One of the main benefits of loop fusion is that it allows temporary memory allocations to be avoided, which can lead to significant performance gains due to lessening memory reading. Other benefits of loop fusion are that it avoids the overhead of the loop control structures. Loop control structures represent significant overhead during a program execution. By fusing loops, the number of control structures needed for the loops is reduces to one, which reduces such an overhead. Besides, when two or more loops are grouped together, it allows the fused-loop body to be computed in parallel by the processor by taking advantages of instruction-level parallelism. One of the most important aspects when fusing loops is to which data each of the separated loops accesses since handling data dependencies among the involved loops to be fused is not a trivial tasks. To illustrate the concept of loop fusion, let us have at the `C++11` code snippet shown in Figure 4.1.

```
for (size_t i = 0; i < N; i++)
{
    x[i] = c;
}
for (size_t i = 0; i < N; i++)
{
    y[i] = f( y[i], x[i]) ;
}
for (size_t i = 0; i < N; i++)
{
    z[i] = g( x[i], y[i]) ;
}
```

(a) Original code with three loops.

```
for (size_t i = 0; i < N; i++)
{
    x[i] = c;
    y[i] = f( y[i], x[i]) ;
    z[i] = g( x[i], y[i]) ;
}
```

(b) Code after loop fusion.

Figure 4.1: Loop fusion example, taken from [8].

In the first for-loop in Figure 4.1 a), all data required to fill in vector `x` are loaded into cache memory from the main memory. If the length of `x`, say N, is such that it does not fit into the cache memory, then as the loop executes its last iterations, it may be required to load into cache the data corresponding to such iterations, while removing the data corresponding to the first iterations, which may imply deleting the latter from the cache memory. When the second loop is executed, data of vector `x` required by the first loop iterations may be reloaded into cache again from the main memory, and we have a similar situation as before. After executing the second loop, the third loop may need to reload data of `x` and `y` into cache. As a consequence of having three separated for loops that share the exact same bounds, data may be copied from the main memory into cache memory three times for vector `x` and twice for vector `y`. Reading data from the main memory is a time-consuming process that should be taken into consideration for performance purposes. Them, the loop fusion technique provides a manner to optimize data usage by replacing the three loops of Figure 4.1 a) with a single loop. This technique may improve data locality since the three operations may reuse data in cache. Consequently, fusing all loops in Figure 4.1 a) into a single loop would be more efficient since data reading would happen once only for each vector and computations would be performed together, assuming that the size of the vectors fits into the cache memory.

It is crucial to notice that loop fusion in Figure 4.1 is possible since all loops share the same bounds and do not include cross-iteration dependencies. This is to bear in mind since a program written in GraphBLAS results in operations that allow parallel execution for different containers, and these operations are performed on containers of the same size, which translates into having loops with the same bounds.

Another key ingredient of the nonblocking execution as proposed in [8] is *loop tiling*, also known as loop blocking or loop partitioning, is a loop optimization technique used to improve cache usage in programs. It involves dividing a loop iteration space into smaller blocks or tiles, and then iterating over these tiles instead of the original loop iterations. In the context of vectors and matrices in ALP/GraphBLAS, the iteration space refers to the length a vector, and the number of rows of a matrix.

The primary concept behind loop tiling is to leverage spatial locality by loading a tile of data into the cache memory and reusing it multiple times before proceeding to the next tile. This approach aids in minimizing cache misses and improving data reuse, which may lead to improved performance. It is important to note that loop tiling introduces additional loop overhead, so the tile size and loop structure should be chosen carefully to achieve the sought performance improvement. Additionally, the effectiveness of loop tiling depends on the characteristics of the system and the memory hierarchy where a program runs.

4.3 In ALP/GraphBLAS

4.3.1 Lazy Evaluation

In programming, when a function is executed or called, it is often the case that this leads to computing certain operations on those variables utilized by the function. This implies that these variables store the actual results of such operations. When the main computations of the function finishes, this returns. Let us take the Code 4.1. In line 1, the function `f` is defined and this is responsible for modifying each element of its input array, line 3. After `f` is called by the `main` function in line 11, it is certain that all elements of the array `arr` have been modified. In this scenario, calling `f` implies that it performs all the computations of which it is in charge. We say then that `f` has been executed.

```
1  f(arr, size) {
2      for ( i = 0; i < size; i++) {
3          arr[i] = i;
4      }
5  }
6
7  main() {
8      arr[10];
9
10     // Call the f function to update the elements of the array
11     f(arr, 10);
12
13 }
```

Code 4.1: Example of function calling and execution.

In GraphBLAS, the blocking execution mode of a primitive corresponds exactly to what is described above i.e. every single primitive in a program is executed immediately after its invocation.

On the other hand, *lazy evaluation* is a programming technique that allows the functions called in a program to delay their computation until their results are actually needed by the program or by the user. By deferring computations, lazy evaluation can help optimize program execution by avoiding unnecessary computations and improving memory usage. However, how to control the correct execution of each function when deferred requires clear understanding of the data dependencies among all the functions.

Due to the fact that ALP/GraphBLAS is a `C++11` implementation, delaying the execution of a primitive is achieved by using a lambda function where the main computations of the primitive are performed.

To achieve nonblocking execution mode in [8], each of the primitives invoked in an ALP/-GraphBLAS program forms a stage of a pipeline. In this context, a primitive being part of a pipeline means that the lambda function of the primitive belongs to the pipeline. We can think of a pipeline as a succession of primitives that are executed sequentially, i.e., the output of one primitive is the input of the next primitive; the order in which each primitive enters the pipeline corresponds to its execution order and this execution order must be followed. From a broad perspective, all primitives that are members of the pipeline share data, which allows levering at run-time the advantages of the loop fusion and loop tiling techniques previously described to improve data usage. From this point on, a primitive belonging to pipeline will be called a stage of that pipeline.

In the nonblocking backend in ALP/GraphBLAS, pipelines are constructed while the program is running, which means that pipelines are built dynamically. A crucial tool to achieve construction of pipelines in an automatic fashion at run-time is a data dependence analysis that decides into which pipeline each primitive is added. Every primitive in the nonblocking backend is implemented so that its main computation is deferred in a lambda function, and then this is added into a pipeline. This implies that each primitive returns without doing the computations for which it is responsible. Whether a primitive forms a stage of a pipeline or another is totally determined by the data this primitive accesses to and what containers it modifies.

We say that a pipeline is executed when all its stages are executed. This means that each of the lambdas in the pipeline does its computations. However, it is of great importance to respect the execution order of each stage to ensure correct results; that is, the first stage added into the pipeline is executed first, then the second one is executed, and so on and so forth.

However, there are cases for which a pipeline must be executed. Such an execution depends upon which the input/output containers, including scalars, involved in the pipeline's stages are. Let us imagine that there is a certain program that leads to having a few pipelines that contain several stages. When a new primitive is to be added into any of these existing, non-empty pipelines, executing any of them in the current implementation of the nonblocking execution mode in ALP/GraphBLAS happens when [8], [11]:

1. the user access to data from a vector or matrix
2. the user creates or destroys a container
3. the user forces the execution of a pipeline by calling the ALP/GraphBLAS function `grb::wait()`
4. the output of the primitive corresponds to a scalar
5. there is a stage in a pipeline that corresponds to a sparse matrix-sparse-vector-multiplication (spM-spV), and the new stage modifies or overwrites the input vector of a spM-spV product. Then, the pipeline containing the spM-spV multiplication is executed, and the new stage is added into an empty pipeline or an existing one

6. there is a stage in the pipelines that corresponds to a sparse matrix-sparse matrix multiplication (spM-spM), and the new stage modifies or overwrites the second input matrix of a spM-spM product. Thus leads to executing the pipeline and adding the new primitive into another pipeline

As a consequence, to avoid prematurely executing a pipeline, from point 1 above, it is advisable to perform all computation within the ALP/GraphBLAS API and to access results in containers once the program has been run. Regarding points 2 and 3, all memory allocations/deallocations should be done before the program is executed, that is, allocating memory at run-time should be avoided whenever possible. Point 4 does not require further discussion.

Point 5 may be unavoidable since it is completely determined by the nature of the program. For example, computing the dot product of two vectors would lead to the execution of the pipeline to which these vectors belong. Likewise, it is quite important to notice that executing a pipeline as described in point 6 above is required due to the spMV algorithm used in the current ALP/GraphBLAS implementation. That is, to compute a single element of the operation $\mathbf{y} = \mathbf{A}\mathbf{x}$, say y_i, the whole vector $\mathbf{x}$ and row $\mathbf{A}(i,:)$ must be available. As a result, if a pipeline contains a spMV operation and the new stage changes its input vector, not executing such a pipeline would result in incorrect computations; the same explanation applies for point 7.

Below it is explained how a pipeline is constructed based on the data dependence analysis that is performed on each primitive.

4.3.2 Parallel computation of tiles

In the nonblocking backend for level-2 primitives that modify vectors, such vectors are split into tiles and each tile is computed in parallel.

Because of this, matrices are split into row-wise tiles. Such tiling consists of dividing a matrix horizontally into smaller, contiguous tiles. Each tile contains a subset of contiguous matrix's rows and all of its columns. The purpose of row tiling is to enable parallel computations on the matrix by distributing the workload across multiple cores or threads. Each tile is allowed to operate on its rows exclusively. Throughout this work, this manner of splitting a matrix into tiles is the only tiling technique matrices. In Figure 4.2, it is shown how tiling is implemented for vectors(right) and matrices(left) in this work.

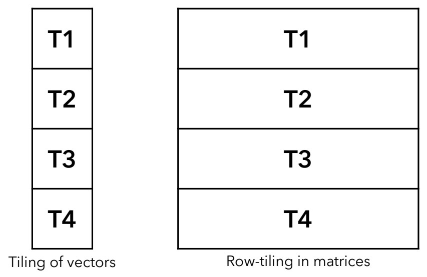


Figure 4.2: Tiling for vectors and matrices in nonblocking backend.

4.3.3 Pipeline construction and data dependence analysis

A key component for the nonblocking execution in [8] is the data dependence analysis that is performed to construct the pipelines at run-time. This process is responsible for adding primitives into pipelines depending upon to which data each primitive accesses. Because of this, while executing a program, an ideal case would be to have pipelines that contained as many stages as possible; this would imply that all stages forming the pipeline shared data and thus there would be better cache usage, i.e. data reading from main memory would be lessened, which translates into a performance improvement.

We now expose the principal ideas behind what pipelines are and how they are constructed. In ALP/GraphBLAS, pipelines are handled using functions in the `pipeline.hpp` file, where the class `Pipeline` exists.

When a ALP/GraphBLAS program is compiled, memory allocations for containers and for pipelines occur. In terms of pipelines, these are stored as elements of array called `pipelines`. In the current implementation, `pipelines` contains four pipelines at compilation time although this may grow depending on the number of pipelines needed during the execution of a program. Each pipeline contains a number of class members; the most relevant ones are detailed below. We refer to the ith pipeline in `pipelines` as Pip$i$.

1. `stages` stores the lambda functions of the stages contained in Pip$i$
2. `opcodes` stores a numerical identifier to label each of the stages contained in Pip$i$
3. `lower_bound` and `upper_bound` store the lower and upper bounds of each of the tiles into which the containers of the stages in Pip$i$ are split
4. `input_vectors` stores the input vectors of the stages in Pip$i$
5. `output_vectors` stores the output vectors of the stages in Pip$i$
6. `vxm_input_vectors` stores the input vectors of the stages that correspond to matrix-vector products in Pip$i$

Additionally, three important functions exist in `pipeline.hpp`: `RC addStage(...)`, which is in charge of adding all the containers and operators needed by a primitive into the corresponding class members of a pipeline, e.g. it adds the name of the primitive into `opcodes` of the pipeline, `RC execution()`, which is in charge of executing the pipeline, and `void merge()`, which merges pipelines into a single one when the data dependence analysis dictates it; (...) indicates that the function takes parameters. It is really important to realize that these functions manipulate or work with pipelines. We refer to these three functions as `pipeline.addStage()`, `pipeline.execution()`, and `pipeline.merge()`.

Similarly, in ALP/GraphBLAS, the data dependence analysis is performed the `lazy_evaluation.hpp` file, where the class `LazyEvaluation` exists; we use `le` to refer to this class. This class also has a method called `RC addStage(...)` whose signature is the same as `pipeline.addStage()`'s. However, the former is responsible for performing the data dependence analysis of a primitive and then adding the primitive into a pipeline. In addition, this class also has a method named `RC execution()`, which simply invokes `pipeline.execution()`. The function `void merge()` also exists in this class and it is invoked by `RC addStage(...)`. As before, to distinguish these methods from `Pipeline`'s, we use `le.addStage()`, `le.execution()`, and `le.merge()`.

To illustrate what has been discussed, let us take the `grb::set(vector, value)` primitive implemented in the nonblocking execution mode in ALP/GraphBLAS. This operation sets each element of `vector` to `value`. After simplifying the code and assuming that all containers are real-valued, we have Code 4.2. We refer to lines in Code 4.2.

```
 1 RC set(vector, val)
 2 {
 3     ...
 4     n = vector.size();
 5     /* Construction of lambda function */
 6     lambda_function = [vector, ...] (pipeline, lower_bound, upper_bound)
 7     {
 8         for( i = lower_bound; i < upper_bound; i++ ) {
 9             vector[ i ] = val;
10         }
11         return SUCCESS;
12     };
13 
14     ret = SUCCESS;
15     /* Add lambda function into a pipeline, containers and associanted data into
           the current pipeline */
16     ret = ret ? ret : le.addStage(
17                 lambda_function,
18                 IO_SET_SCALAR,
19                 n, ...,
20                 vector, ...);
21     ...
22     return ret;
23 }
```

Code 4.2: Implementation of `grb::set()` in ALP/GraphBLAS, nonblocking mode.

The function shown in Code 4.2 clearly highlights all important elements of an ALP/GraphBLAS primitive implemented in the nonblocking backend, and how the execution of a function can be delayed by using lambda functions. In line 1, we observe the return type `RC` of the `grb::set` primitive, which is the same for most of the ALP/GraphBLAS primitives.

Indeed, the main computations of this primitive can be simply implemented by using a single for-loop over the total elements of `vector`. In line 4, the size of `vector` is retrieved. Afterwards, in line 6 we encounter a lambda function where the main operation for which `grb::set` is responsible occurs. Since this lambda function must be added into a pipeline, it takes a pipeline as a parameter. Moreover, it takes a lower and upper bound parameters; these correspond to the tile bounds within which `vector` is modified by this lambda.

Then, in line 7, it is encountered a for-loop where each element of `vector` is set to `val`. However, we notice that `grb::set` does not perform that loop; it is stored inside the lambda function.

After defining the lambda function, this is passed to the `le.addStage()` function, which analyzes the input and output containers of the stages to which it corresponds, line 17. In line 24, we observe that the returning value of `grb::set` is the error code thrown by the`le.addStage()` function; the error code returned by `grb::set` is not related to whether or not the actual computations are properly done.

The `grb::set` primitive highlights well how each ALP/GraphBLAS primitive is essentially structured in the nonblocking execution mode: each primitive has a main task to performed, which is writeen inside a lambda function, and then each lambda function is added into a pipeline based on a data dependence analysis performed by the function `le.addStage()`. Apart from specific cases, none of the primitives in the nonblocking mode actually performs their tasks.

In order to detail what the previous methods from the `Pipelines` and `le` classes do, we now explain the idea of data dependence analysis and pipeline construction by using specific examples. Herein the data dependence analysis is explained for level-3 primitives; for level-2, the same ideas can be applied. This data dependence analysis is done automatically inside the `le.addStage()` method, which is called at the end of every single primitive implemented in the nonblocking backend.

Throughout this work, the array `pipelines` is graphically represented as shown in Figure 4.3, where Pip1 and Pip2 are the pipelines' names, and stages that belong to each pipeline are added below the pipeline's name. For simplicity, only Pip1 and Pip2 are displayed. Below each pipeline's names, we encounter its stages; the stage right under the pipeline's name is the first to be added into the pipeline, and so on and so forth.

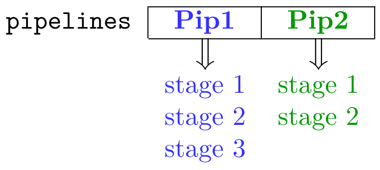


Figure 4.3: Convention for representing the array `pipelines`.

For all cases below, we describe what would happen during the execution of a program in terms of pipelines if the state of `pipelines` were given. We achieve this by assuming that there is a current state of `pipelines` for each case and that there is a new primitive to be added into `pipelines`; we refer to it as the new primitive. Importantly, it should be reminded that each primitive corresponds to a mathematical operation.

The data dependence analysis shown below is an implementation this work that uses similar ideas to the data dependence that is implemented for level-2 primitives; therefore, this data dependence analysis also applies for level-2 primitives.

To begin with, the trivial case in the data dependence analysis corresponds to the case when all pipelines are empty and there is a new primitive, say the one corresponding to the operation $\mathbf{C} = \mathbf{AB}$, to be included into `pipelines`. When `le.addStage()` is called at the end of the primitive, no data dependence analysis is performed since there are no primitives in `pipelines`. Under such circumstances, the primitive is added as the first stage of Pip1.

Correspondingly, one of the simplest cases in the data dependence analysis happens when the operation $\mathbf{R} = \mathbf{P} + \mathbf{Q}$ exists in Pip1, the new operation is $\mathbf{C} = \mathbf{AB}$, and the remaining pipelines are empty. We observe that both operations access to different data and then we say that they share no data, the latter operation is added into Pip2 as its first stage, Figure 4.4.

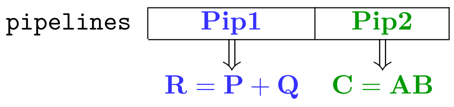

pipelines
Pip1
Pip2
$\mathbf{R} = \mathbf{P} + \mathbf{Q}$
$\mathbf{C} = \mathbf{AB}$

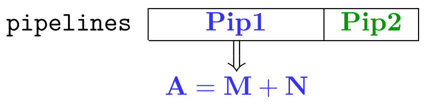
Figure 4.4: Operations share no data.

Let us now imagine that in a given program the operation $\mathbf{A} = \mathbf{M} + \mathbf{N}$ is already the first stage of Pip1, that the new operation is $\mathbf{C} = \mathbf{A} + \mathbf{B}$, and that all other pipelines are empty, see Figure (4.5).

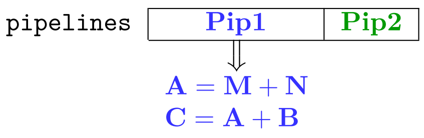
Moreover, since tiling of the output matrices $\mathbf{A}$ and $\mathbf{C}$ is done in a row-wise fashion, each of their tiles would be schematically computed as shown in Figure 4.6, where four threads are assumed to be available for computations and each corresponds to a tile; for example, thread 1 corresponds to tile 1. We focus on tile T1.

Each tile works on a specific part of the input matrix and its limits are determined...

From Figure 4.6, if tile T1 finishes computing what it is responsible for in $\mathbf{A} = \mathbf{M} + \mathbf{N}$, Figure 4.6a, then it will immediately go computing its part in $\mathbf{C} = \mathbf{A} + \mathbf{B}$, Figure 4.6b, without having to wait for the other tiles to finish. From a data access perspective, this means that T1

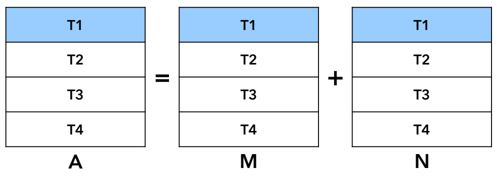


(a) Pip1 contains a single operation.

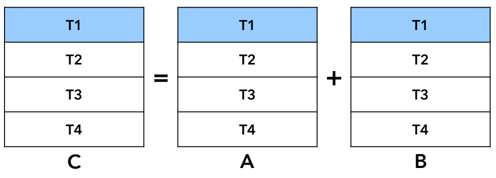


(b) The second operation forms the second stage of Pip1.

Figure 4.5: Operations share data, thus they belong to same pipeline Pip1.



(a) First stage of Pip1.



(b) Operation to be added into a pipeline.

Figure 4.6: Matrix tiling for two operation in Pip1.

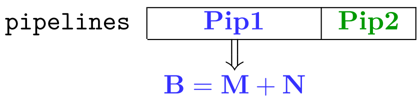
loads into its memory the corresponding parts of matrices $\mathbf{M}$ and $\mathbf{N}$ for the first operation, and then it reads what data it needs from $\mathbf{A}$ and $\mathbf{B}$ for the second operation. This highlights the fact that the same part of $\mathbf{A}$ may be read twice from the main memory by the same thread. Thus, we state that both operations share data and, consequently, $\mathbf{C} = \mathbf{A} + \mathbf{B}$ is added as the second stage of Pip1, Figure 4.5b. Furthermore, since both operations belong to Pip1, such a pipeline contains the lambda functions, input matrices, output matrices, and operation's names corresponding to $\mathbf{A} = \mathbf{M} + \mathbf{N}$ and $\mathbf{C} = \mathbf{A} + \mathbf{B}$. The same conclusions would follow if we had the operations $\mathbf{B} = \mathbf{M} + \mathbf{N}$ and $\mathbf{C} = \mathbf{A} + \mathbf{B}$, as the existing operation in Pip1 and the operation to be added into `pipelines`, respectively.

Similarly, let us imagine that Pip1 contains $\mathbf{A} = \mathbf{MN}$ as its first stage and that the operation $\mathbf{C} = \mathbf{A} + \mathbf{B}$ is to be added into `pipelines`. By employing a similar line of reasoning as in the previous case above, we can deduce that $\mathbf{C} = \mathbf{A} + \mathbf{B}$ would form the second stage of Pip1. The same occurs when $\mathbf{C} = \mathbf{M} + \mathbf{N}$ is already Pip1 and the operation $\mathbf{C} = \mathbf{AB}$ is to be added into a pipeline.

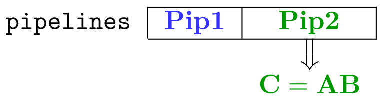
Now if the operation $\mathbf{B} = \mathbf{M} + \mathbf{N}$ is already in Pip1, and the operation $\mathbf{C} = \mathbf{AB}$ is to be added into `pipelines`, we observe that these operations, indeed, share data. Nevertheless, to compute any tile of $\mathbf{C}$, the whole matrix $\mathbf{B}$ may be required, see Figure 4.7b, which is modified by the existing operation in Pip1. Consequently, Pip1 is executed and the primitive of $\mathbf{C} = \mathbf{AB}$ is added as the first stage of another empty pipeline, i.e., Pip2, Figure 4.7. At this point, $\mathbf{B}$ has been fully computed and Pip1 is again an empty pipeline. A similar reasoning would apply if the existing operation in Pip1 were $\mathbf{B} = \mathbf{MN}$.

Likewise, having the operation $\mathbf{M} = \mathbf{NC}$ as the existing one in Pip1 and the operation $\mathbf{C} = \mathbf{AB}$ as the one to be added into `pipelines` would lead to executing Pip1 and adding the second operation as the first stage of Pip2 since the latter modifies one of the input matrices of the former.

The previous case has a level-2 analogous. For vectors $\mathbf{a}, \mathbf{b}, \mathbf{x}, \mathbf{y} \in \mathrm{R}^n$ and matrix $\mathbf{A} \in \mathrm{R}^{n \times n}$, if the operation in Pip1 is $\mathbf{x} = \mathbf{a} + \mathbf{b}$ and the operation to be added into `pipelines` is the product $\mathbf{y} = \mathbf{Ax}$, then to compute a single element of $\mathbf{y}$, say y_i, all the elements of vector $\mathbf{x}$ are needed. Consequently, Pip1 would be executed and $\mathbf{y} = \mathbf{Ax}$ would be added into Pip2.

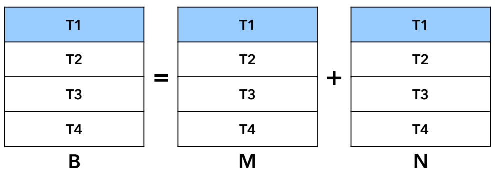


(a) Pip1 contains $\mathbf{B} = \mathbf{M} + \mathbf{N}$.

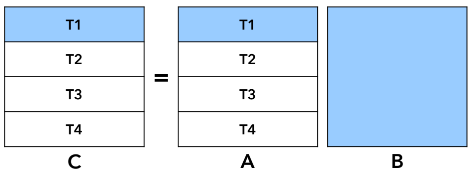


(b) `pipelines` after adding the second operation to Pip2.

Figure 4.7: Pip1 contains an operation that overwrites the second input matrix of $\mathbf{C} = \mathbf{AB}$, which leads to executing Pip1.



(a) First stage of Pip1.

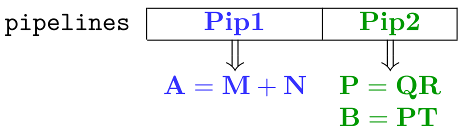


(b) Operation to be added into a pipeline.

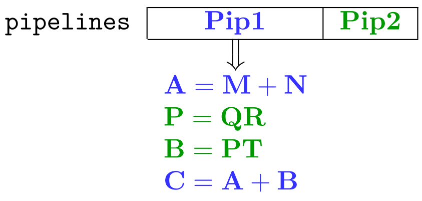
Figure 4.8: Matrix tiling for two operation in Pip1.

From the previous case, we notice that if the operation to be added into `pipelines` is a spM-spM multiplication and there is a pipeline that contains an operation that modifies the second input matrix of that multiplication, the pipeline is executed and the new operation is added as the first stage of an another pipeline. In similar fashion, if both the existing operation in the pipeline and the one to be added into `pipelines` are spM-spM multiplications, then if the latter modifies the second input of the former, the pipeline is executed and the new operation goes as the first stage of an another pipeline.

Now let us say that Pip1 contains the operation $\mathbf{A} = \mathbf{M} + \mathbf{N}$, and Pip2 has $\mathbf{P} = \mathbf{QR}$ and $\mathbf{B} = \mathbf{PQ}$ as its first and second stages, respectively, and the operation to be added into `pipelines` is $\mathbf{C} = \mathbf{A} + \mathbf{B}$, Figure 4.9a. From what has been explained before, it is observed that $\mathbf{C} = \mathbf{A} + \mathbf{B}$ shares data with Pip1 and Pip2. Then both Pip1 and Pip2 are merged into a single pipeline; that is, stages in Pip2 are moved into Pip1's and the new operation is added as the last step of the just-merged pipeline Pip1. After the merging step, Pip2 is an empty pipeline, Figure 4.9b.



(a) Pip1 and Pip2 contain stages that share data.



(b) Pip1 after merging Pip2's stages.

Figure 4.9: Stages in Pip1 and Pip2 share data, which leads to merging both pipelines.

In Figure 4.9b, the coloring for stages that were part of Pip2 is kept as it was before merging to highlight where they are in Pip1. The most important aspect when merging two or more pipelines is to keep the relative order of operations in each pipeline. From Figure 4.9b, all stages in Pip1 that belonged to Pip2 follow exactly the same order they had when they were part of

Pip2.

Moreover, if Pip1 has an operation like $\mathbf{C} = \mathbf{A}+\mathbf{B}$ and the operation $\mathbf{C} = \mathbf{MN}$ is to be added into `pipelines`, there is shared data between both operations; however, both could change the sparsity pattern of $\mathbf{C}$. Thus, Pip1 execution occurs and $\mathbf{C} = \mathbf{MN}$ goes to another empty pipeline or to a pipeline with with it shares data for which the output of its stages is not $\mathbf{C}$. This case illustrates that in the current nonblocking implementation of ALP/GraphBLAS it is not allowed having two primitives that modify the sparsity of the same output matrix in the same pipeline.

For all of the previous cases, a pipeline is executed if the output of the operation to be added into `pipelines` is a scalar since scalars are not implemented as opaque data types in ALP/GraphBLAS.

From the previous discussion, data dependence analysis constructs pipelines based on what data each of the primitives called during the program execution accesses to. It is important to state that the spM-spM and spM-spV products could lead to executing a pipeline as shown previously. However, the most important idea to bear in mind is that such operations lead to execution of a pipeline because of what algorithms are implemented to compute each of them. For example, the matrix-matrix product is computed using the row-wise Gustavson scheme, which requires the whole matrix $\mathbf{B}$ to compute a single row of $\mathbf{C} = \mathbf{AB}$.

4.3.4 Coordinates mechanism for sparse vectors

Vectors in ALP/GraphBLAS may be either sparse or dense. When a vector is dense, each primitive using the vector as a parameter accesses all its elements. On the other hand, to efficiently handle sparse vectors, it is necessary to maintain the coordinates of its nonzero elements so that ALP/GraphBLAS operations access the nonzero values exclusively. Such coordinates are used to access the nonzero elements of sparse vectors. Therefore, for each sparse vector, a *Sparse Accumulator*(SPA), explained in detail in chapter 5, is created. In ALP/GraphBLAS, the SPA of any vector is accessed to and modified by the `Coordinates` class declared in the `coordinates.hpp` file. All functions required to update the sparsity information of a vector are encountered in that class. For any sparse vector, its SPA is referred to as its coordinates. A `coordinates` object has a few important member variables shown below.

- An unsigned integer `cap` that stores the length the vector
- An unsigned integer `n` that stores the number of nonzero values in the vector
- A boolean array, `assigned`, of length `cap` that indicates if the ith element of the vector is a nonzero. That is, `assigned[i]` is true whenever the ith element of the vector is a nonzero
- An unsigned integer array, `stack`, that stores the indices of the assigned elements, i.e., if `assigned[i]` is true, then `stack[j] = i`

We notice that `stack` and the `assigned` arrays are used only when working with a sparse vector. For an empty vector, `n=0`, all the elements of `assigned` are initialised to false, and `stack` is empty. When the ith element of the `v` is assigned, this implies that `stack[n]=i`, `assigned[i]=T`, `n=1`. That is, the index of the nonzero is pushed into the `stack`, the corresponding element in the `assigned` array is set to true and the number of nonzeros in the vector increases by one. It is important to state that indices of the nonzeros are not sorted in `stack` - they are pushed in an arbitrary order-, and that iterating over the nonzeros of a sparse vector is done through `stack`; then access to the nonzero elements may happen in any order.

To illustrate the importance of the coordinates of a sparse vector, let us imagine that in a program we have a primitive that changes the sparsity of the sparse vector `v` whose original state is such that `cap=10` and `n=5`, see Figure 4.10 (top), where T and F stand for true and false, respectively. That primitive is such that it adds two new nonzeros, each equal to 1, into `v[i]` and `v[j]` where `i=1` and `j=7`.

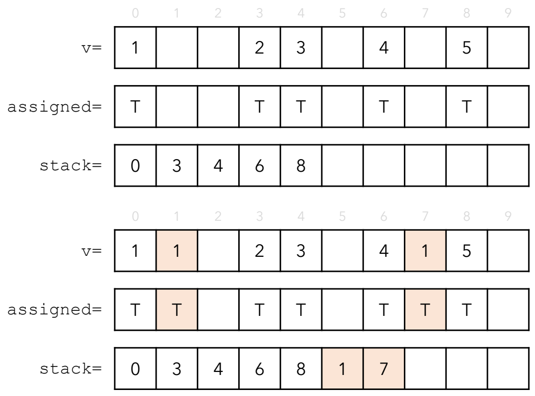


Figure 4.10: Sparse vector `v` that is updated in the reference backend.

In the reference backend all computations are performed sequentially; that is, after calling that primitive `assigned[i]=T`, `assigned[j]=T`, `stack[5]=i`, `stack[6]=j`, and `n=7`, see Figure (bottom). The coordinate mechanism of sparse vectors correctly manages updating the sparsity of `v`: it first updates `stack[5]=i` and `n=6`, and then it assigns `stack[6]=j` and `n=7`; sequential execution implies that one element of the coordinates of `v` is updated at a time. However, this is not the case for parallel execution in the nonblocking backend.

Concerning the same case as before in the nonblocking backend, let us imagine that `v` is updated by using two tiles, one that is responsible for `v[0:4]` and the other for `v[5:9]`, see Figure 4.10. Updating `assigned` in parallel causes no data races, which means that the first tile performing `assigned[i]=T` at the same time the other tile performing `assigned[j]=T` is correct as long as `i` and `j` are distinct. The same reasoning applies to updating `v[i]` and `v[j]` in parallel. Nevertheless, there are data races when both tiles try modifying `stack` and `n`: both tiles could be trying to write into `stack` and to increase `n` simultaneously.

Therefore, the coordinate mechanism for sparse vector is extended such that it correctly updates of `stack` and `n` in parallel nonblocking execution. This is what is called the *local coordinates mechanism* for sparse vectors. This is one of the core ideas presented in [8] that enables the nonblocking execution of level-2 primitives. In this case, the coordinates of the entire vector `v` are called the *global view* of `v`. The local coordinates mechanism consists of creating a set of local copies or views of the global view of `v`. There are as many local views as number of threads and each thread handles its own coordinates. For example, let us say that for a program to be run in the nonblocking backend, there a few primitives forming the stages of a pipeline and these access to and modify the sparse vector `v` in Figure 4.11. When that pipeline is executed, such primitives do not work with the global view of `v`; instead, they work with and modify its local views. After the execution of the pipeline, those local views are used to update the global view of `v`.

The usage of the local views is mainly justified by the fact that data races exist when modifying in parallel the `stack` array and the `n` of the coordinates of a sparse vector.

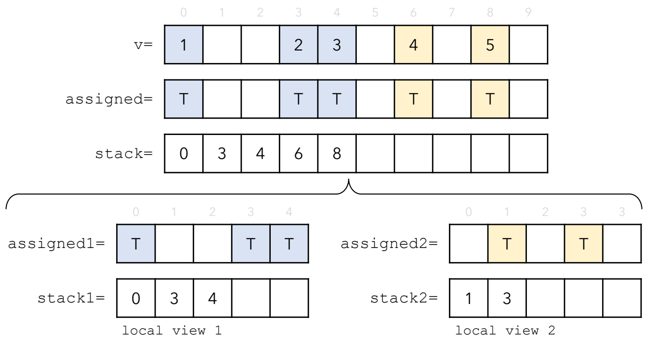


Figure 4.11: Local views of the sparse vector `v`.

4.3.5 Conjugate gradient using nonblocking execution

The ideas described for data dependence analysis in section 4.3.3 can be applied similarly to level-2 operations. Therefore, herein we discuss how the Conjugate Gradient (CG) algorithm is executed in the nonblocking mode of ALP/GraphBLAS. This can be found in [4] as ALGORITHM 11.15.

Problem statement for applying the CG method. For a known symmetric, positive-definite matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, and a known vector $\mathbf{b} \in \mathbb{R}^{n}$, find the solution vector $\mathbf{x} \in \mathbb{R}^{n}$ to system of linear equations of the form

$$\mathbf{b} = \mathbf{A}\mathbf{x}. \tag{3.1}$$

The CG algorithm as presented in [4] is written in Algorithm 1.

On the other hand, Algorithm 1 can be written in ALP/GraphBLAS as shown in Code 4.3. From this point and on, we refer to lines of Code 4.3.

To represent the current state of the array `pipelines`, the convention shown in Figure 4.12 will be employed to to identify each stage.

line ALP/GraphBLAS primitive | mathematical notation of primitive

Figure 4.12: Convention for stages in pipelines.

In Figure 4.12, line refers to the line of code in code 4.3, and we notice that the CG method works mainly with matrices and vectors; thus most of the primitives mentioned below work with level-2 and level-3 containers. It is explicitly stated when a primitive outputs a scalar.

At initialization of the program execution, the array `pipelines` stores up to four pipelines, all of which are empty. We observe that in line 15, two scalars are defined; these are not added into any pipeline. To begin adding stages into `pipelines`, we notice that the first primitive that can actually be pipelined during the program execution corresponds to the ALP/GraphBLAS primitive `grb::set(temp, 0)`, line 18. Since this is the very first primitive to be added into `pipelines`, such a primitive is added as first stage of the first pipeline Pip1; the array `pipelines` looks like Figure 4.13.

The second primitive to be added into `pipelines` is encountered in line 21, grb::set(r, 0)|r=0; we notice that such a primitive has no data sharing with grb::set(temp, 0)|temp =0 since each primitive accesses different containers - both of them have distinct outputs; then, line 21 is added into an empty pipeline, Pip2, which leads to having Figure 4.14.

Algorithm 1: Conjugate Gradient (CG) algorithm

```
Input  : maximum number of iterations max_iter
         tolerance
         $\mathbf{r}_0 \coloneqq \mathbf{b} - \mathbf{A}\mathbf{x}_0$
         if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result
         $\mathbf{p}_0 \coloneqq \mathbf{r}_0$
         $k \coloneqq 0$
Output: Solution $\mathbf{x}$ to $\mathbf{b} = \mathbf{A}\mathbf{x}$
1  while $max\_iter \leq k$ do
2      $$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k} \tag{3.2}$$
3      $$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{3.3}$$
4      $$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k \tag{3.4}$$
5      if $\mathbf{r}_{k+1} < tolerance \cdot tolerance$ then
6          return $\mathbf{x}_{k+1}$ as the result
7      end
8      $$\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k} \tag{3.5}$$
9      $$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \tag{3.6}$$
10     $k = k + 1$
11 end
12 return $\mathbf{x}_{k+1}$ as the result
```

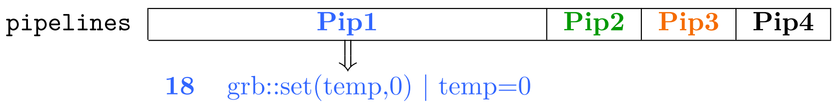


Figure 4.13: Adding primitive `grb::set(temp, 0)` into Pip1.

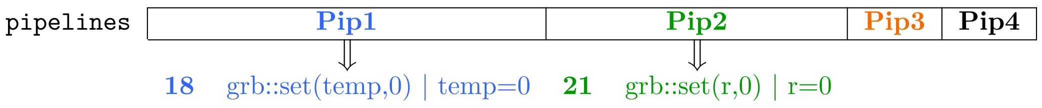


Figure 4.14: Adding primitive `grb::set(r, 0)` into a pipeline.

Regarding line 24, grb::mxv(temp, A, x, ring)|temp = Ax, the container temp is the output of this primitive. From the non-empty pipelines Pip1 and Pip2, we observe that the output of the first primitive in Pip1, line 18, corresponds to temp too; such a fact indicates that this primitive shares data with the first primitive in Pip1. As a result, `grb::mxv(temp, A, x, ring)` is added as the second stage of Pip1; we thus have Figure 4.15 .

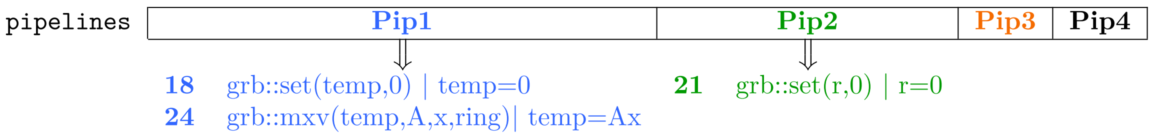

pipelines
Pip1
Pip2
Pip3
Pip4
18 grb::set(temp,0) | temp=0
24 grb::mxv(temp,A,x,ring)| temp=Ax
21 grb::set(r,0) | r=0

Figure 4.15: Adding primitive `grb::set(r, 0)` into Pip2.

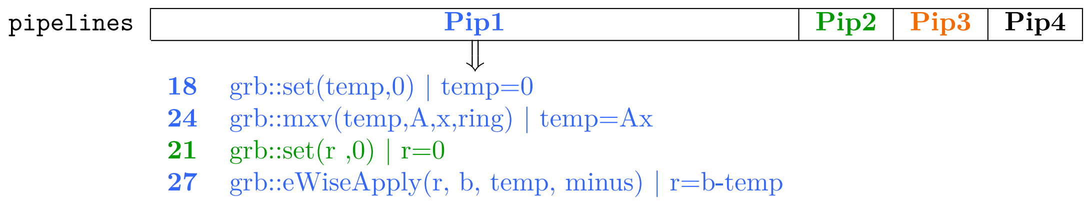
The next primitive in the program is grb::eWiseApply(r, b, temp, minus)|r = b - temp, line 27. Firstly, this primitive accesses the container temp, which is also a container in the first stage of Pip1. Secondly, the same primitive accesses the container r, which is found as a container in the first stage of Pip2. As a consequence, the primitive in line 27 shares data with primitives in both pipelines Pip1 and Pip2. Thus, Pip1 and Pip2 are merged together into a single pipeline, which means they form a single pipeline, the stages of which are the stages in Pip1 and Pip2. It is important to mention that the order of the stages in each pipeline to be merged must be kept. That is, in the resulting merged pipeline, the first stage in Pip1 must proceed its second one.

Similarly, merging Pip1 and Pip2 can be done in any order; that is, the stages in Pip1 can be added into the new merged pipeline before the ones in Pip2 or vice-versa, as long as the relative order of stages in each pipeline is respected, see Figure 4.17. The current implementation of ALP/GraphBLAS merges Pip2 into Pip1, which results in turning Pip2 into an empty pipeline. After merging both pipelines, the primitive in line 27 is added as the last stage of the just-merged pipeline. Then we have Figure 4.16, where the coloring of stages in Pip2 is kept as it was before merging both pipelines as a manner to emphasize which stages corresponded to Pip2.

In Figure 4.17, as an example, it is shown an equivalent pipeline to the one shown in Figure 4.16. Being equivalent in this context means that the execution of either of these pipelines would lead to the same results. We notice that after merging both pipelines we have a single non-empty pipeline, Pip1, and, henceforward, we work with `pipelines` shown in Figure 4.16.

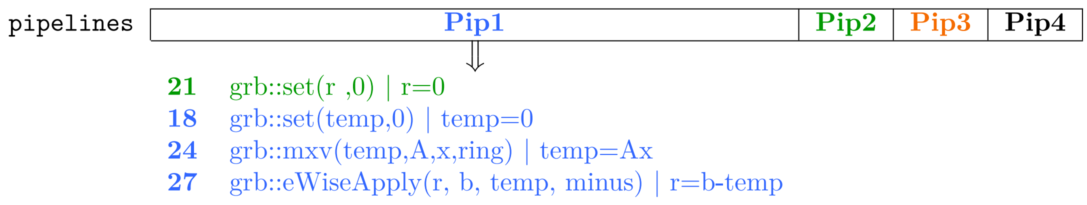
Now, a new stage is to be added into `pipelines`. Similar arguments as before show that the primitive `grb::set(u, r)`, in line 30, shares data with Pip1, which means that such a primitive is added as the last stage of Pip1, Figure 4.18.

Likewise, to add primitive `grb::dot(sigma, r, r, ring)`, line 33, into a pipeline, we notice that, indeed, it shares data with pipeline Pip1 in `pipelines`; thus, this is added as the last stage



18 grb::set(temp,0) | temp=0
24 grb::mxv(temp,A,x,ring) | temp=Ax
21 grb::set(r ,0) | r=0
27 grb::eWiseApply(r, b, temp, minus) | r=b-temp

Figure 4.16: Adding primitive `grb::eWiseApply(r, b, temp, minus)` into a pipeline.



21 grb::set(r ,0) | r=0
18 grb::set(temp,0) | temp=0
24 grb::mxv(temp,A,x,ring) | temp=Ax
27 grb::eWiseApply(r, b, temp, minus) | r=b-temp

Figure 4.17: Adding primitive `grb::eWiseApply(r, b, temp, minus)` into a pipeline, equivalent to Figure 4.16.

of Pip1. We have Figure 4.19.

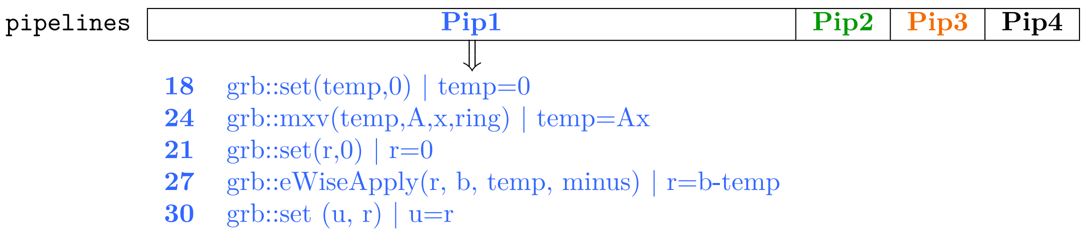
Since the output of `grb::dot(sigma, r, r, ring)` corresponds to a scalar, after adding such a primitive into Pip1, Pip1 is executed. At this point, we recall that the matrices and vectors are tiled and each tiles is performed in parallel. In [8], an analytic model is used to determine automatically for each pipeline at run-time the tile size and the number of tiles based on L1 cache size and the number of cores available on the system. The crucial idea to remember when Pip1 is executed is that each tile receives the set of lambda functions to execute that correspond to the primitives in the pipeline, and since each tile is aware of what its bounds are, each tile modifies a specific part of the containers accessed by the primitives that form the pipeline. Moreover, each tile computes its primitives independently of what the others tiles do. For example, let us say that tile T1 finishes executing the first primitive in the pipeline, then it continues computing the second operation without waiting on the remaining tiles, and so on and so forth. It is important to notice that to compute the primitive `grb::mxv(temp, A, x, ring)`, vector `x` must be fully available

It is important to mention that after the execution of Pip1, all of its primitives are completely executed; that means that each of the containers accessed by the primitives in Pip1 contains their actual computations. Likewise, Pip1 is fully executed before the program moves on to next code lines. For instance, using the same example as before, let us imagine that tile T1 finishes computing all of its tasks and no more tasks are to be performed in Pip1, then T1 does not perform any computation. After Pip1 has been executed, its elements are cleared, and the program resumes its execution.

Then, regarding line 36, this corresponds to a scalar and then it is not added into `pipelines`.

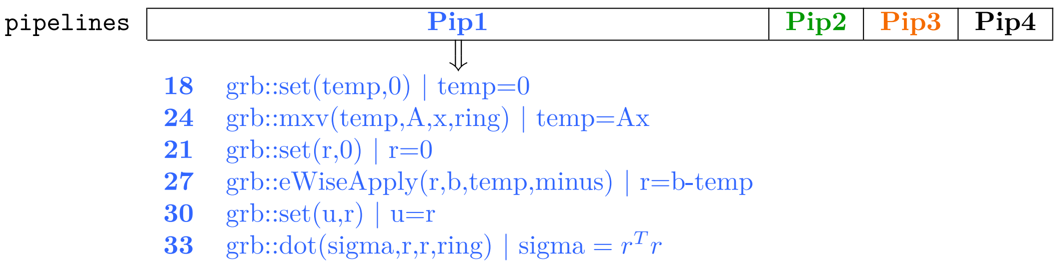
Now, the program execution enters the do-while loop in line 38, which corresponds to the main computations of the CG algorithm. We detail how the pipelining of primitives during a single iteration of that loop is performed.

At this point of the program execution, `pipelines` contains empty pipelines. To add the primitive grb::mxv(temp, A, u, ring)|temp = Au, line 40, since all pipelines are empty, this primitive is added as the first stage of Pip1. Similarly, grb::dot(residual, temp, u, ring)|residual = temp^{T}u, line 43, shares data with the first stage in Pip1, and thus it is added as the second stage in Pip2, see Figure 4.20.



18 grb::set(temp,0) | temp=0
24 grb::mxv(temp,A,x,ring) | temp=Ax
21 grb::set(r,0) | r=0
27 grb::eWiseApply(r, b, temp, minus) | r=b-temp
30 grb::set (u, r) | u=r

Figure 4.18: Adding primitive `grb::set(u, r)` into a Pip1.



18 grb::set(temp,0) | temp=0
24 grb::mxv(temp,A,x,ring) | temp=Ax
21 grb::set(r,0) | r=0
27 grb::eWiseApply(r,b,temp,minus) | r=b-temp
30 grb::set(u,r) | u=r
33 grb::dot(sigma,r,r,ring) | sigma = $r^T r$

Figure 4.19: Adding primitive `grb::dot(sigma, r, r, ring)` into a pipeline.

As before, after adding `grb::dot(residual, temp, u, ring)` into Pip1, such a pipeline is executed since the output of the that primitive is a scalar value. This implies having all pipelines empty.

The primitive `grb::apply(alpha, sigma, residual, divide)`, line 46, is an operation on scalars only; this is not added into `pipelines`.

On the other hand, the primitive `grb::eWiseMulAdd(x, alpha, u, x, ring)`, line 49, is added as the first stage of Pip1 since all pipelines are empty. Because the next primitive to be added into `pipelines`, `grb::eWiseMul(temp, alpha, temp, ring)`, line 52, shares no data with Pip1, this forms the first stage of Pip2. Likewise, both `grb::eWiseApply(r, r, temp, minus)`, line 55, and `grb::dot(residual, r, r, ring)`, line 58 share data with Pip2 and, thus are added as the second and third stages of Pip2, respectively; see Figure 4.21.

As the output of the last stage in Pip2 is a scalar, Pip2 is executed, which leaves `pipelines` with a single non-empty pipeline Pip1. Let us assume that the condition in the if statement in line 61 is false. Then, grb::apply(alpha, residual, sigma, divide), line 66, is executed immediately since it is a scalar. The primitive `grb::eWiseMulAdd(u, alpha, u, r, ring)`, line 69, is added as the second stage of Pip1 because it shares data with the first stage of Pip1. Primitive in line 72, sigma = residual, is a scalar. At this line, the first iteration of the do-while loop, starting at line 38, finishes and the state of `pipelines` is shown in Figure 4.22.

When the second iteration of the do-while loop starts, using the same reasoning as before, the primitives in lines 40 and 43 share data with Pip1, then these are added as the third and fourth stages of Pip1. After adding the latter, Pip1 is executed since such a primitive corresponds to a scalar.

The processes of adding stages into `pipelines` and executing pipelines when required is repeated until either the tolerance is reached or the total number of iterations reaches it maximum value.

```
1     grb::RC conjugate_gradient( x,
2         A,
3         b,
```

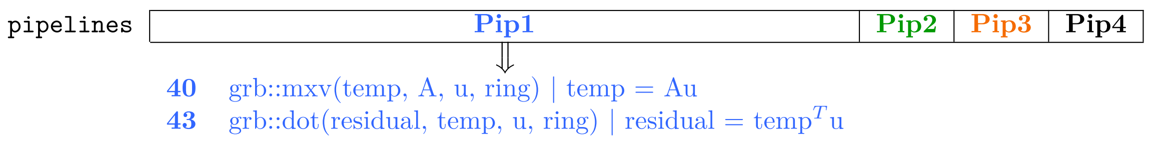

pipelines
Pip1
Pip2
Pip3
Pip4
40 grb::mxv(temp, A, u, ring) | temp = Au
43 grb::dot(residual, temp, u, ring) | residual = temp^{T}u

Figure 4.20: Adding `grb::mxv(temp, A, u, ring)` and `grb::dot(residual, temp, u, ring)` into Pip1.

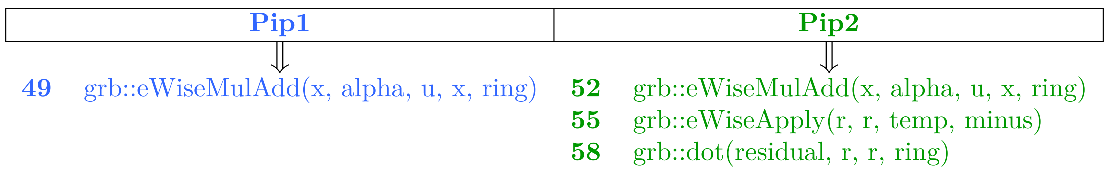

Pip1
Pip2
49 grb::eWiseMulAdd(x, alpha, u, x, ring)
52 grb::eWiseMulAdd(x, alpha, u, x, ring)
55 grb::eWiseApply(r, r, temp, minus)
58 grb::dot(residual, r, r, ring)

Figure 4.21: Current state of `pipelines` after adding primitives in lines 49, 52, 55, and 58.

```
4           max_iterations,
5           tol,
6           iterations,
7           residual,
8           r,
9           u,
10          temp,
11          ring,
12          minus,
13          divide ) {
14
15          alpha, sigma;
16
17          /* operation: temp = 0 */
18          grb::set( temp, 0 );
19
20          /* operation: r = 0 */
21          grb::set( r, 0 );
22
23          /* operation: temp = A * x */
24          grb::mxv( temp, A, x, ring );
25
26          /* operation: r = b - temp */
27          grb::eWiseApply( r, b, temp, minus );
28
29          /* operation: u = r */
30          grb::set( u, r );
31
32          /* operation: sigma = r' * r */
33          grb::dot( sigma, r, r, ring ); //
34
35          /* operation: iter = 0 */
36          iter = 0;
37
38          do {
39              /* operation: temp = A * u */
40              grb::mxv( temp, A, u, ring );
41
42              /* operation: residual = u' * temp */
43              grb::dot( residual, temp, u, ring );
44
```

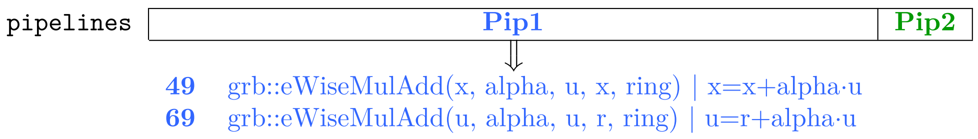

pipelines
Pip1
Pip2
49 grb::eWiseMulAdd(x, alpha, u, x, ring) | x=x+alpha·u
69 grb::eWiseMulAdd(u, alpha, u, r, ring) | u=r+alpha·u

Figure 4.22: Current state of `pipelines` after adding primitive in line 69.

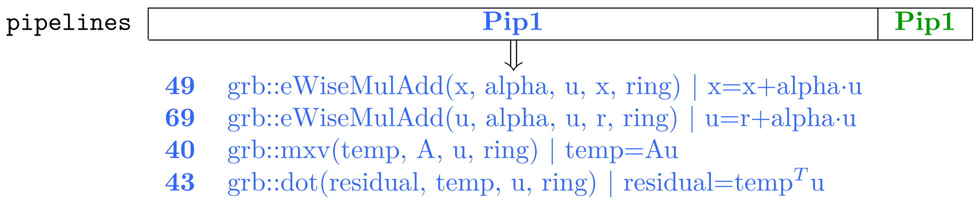

pipelines
Pip1
Pip1
49 grb::eWiseMulAdd(x, alpha, u, x, ring) | x=x+alpha·u
69 grb::eWiseMulAdd(u, alpha, u, r, ring) | u=r+alpha·u
40 grb::mxv(temp, A, u, ring) | temp=Au
43 grb::dot(residual, temp, u, ring) | residual=temp^T u

Figure 4.23: `pipelines` after adding primitive in lines 40 and 43, second do-while loop iteration.

```
45             /* operation: alpha = sigma / residual */
46             grb::apply( alpha, sigma, residual, divide );
47 
48             /* operation:  x = x + alpha * u */
49             grb::eWiseMulAdd( x, alpha, u, x, ring );
50 
51             /* operation: temp = alpha * temp */
52             grb::eWiseMul( temp, alpha, temp, ring );
53 
54             /* operation: r = r - temp */
55             grb::eWiseApply( r, r, temp, minus );
56 
57             /* operation: residual = r' * r */
58             grb::dot( residual, r, r, ring );
59 
60             /* we check if residual is smaller than chosen tolerance  */
61             if( residual < tol * tol ) {
62                 break;
63             }
64 
65             /* operation: alpha = residual / sigma */
66             grb::apply( alpha, residual, sigma,divide );
67 
68             /* operation: u = r + alpha * u */
69             grb::eWiseMulAdd( u, alpha, u, r, ring );
70 
71             /* operation: sigma = residual */
72             sigma = residual;
73 
74             } while( iter++ < max_iterations );
75 
76             // output
77             iterations = iter;
78 
79             return SUCCESS;
80     }
```
Code 4.3: CG algorithm implemented in ALP/GraphBLAS

Chapter 5

Sparse matrix - sparse matrix (spM-spM) multiplication

5.1 CRS and CCS formats and SPA data structure

The most widely used storage formats to store sparse matrices are the *Compressed Sparse Column* (CSC) and the *Compressed Sparse Row* (CRS) formats, [6]. In the CRS format, the sparse matrix is stored row-wise, i.e., each row is stored contiguously. The nonzero values of the matrix are stored in a single, dense array, and the corresponding column indices of these nonzeros are stored in another array. Additionally, another array is used to store the starting index of each row in the arrays that store the value and column indices. The CRS format allows for efficient access of matrix elements by rows.

On the other hand, the CCS format, the sparse matrix is stored column-wise. The nonzero values of the matrix are stored in a single, dense array, and the corresponding row indices of these values are stored in another array. Similar to the CRS format, an additional array is used to store the starting index of each column in the value and row index arrays. This format enables efficient access of matrix elements by columns. This format enables efficient access of matrix elements by columns.

These two formats provide a compact representation for sparse matrices on computers, where only the nonzeros are stored, resulting in less memory usage compared to dense matrices.

In ALP/GraphBLAS, the class `Matrix` is in charge of initializing any matrix, which includes allocating and deallocating memory, resizing a matrix, and so on. Moreover, each matrix has several member variables like its number of columns and rows, and its capacity. When we instantiate an object from such a class, a matrix is created. In ALP/GraphBLAS, matrices are stored using the Gustavson format [11]; that is, each matrix is stored twice, once in the CRS format and another in the CCS format. We provide a few important details about the CRS format as implemented in the reference backend of ALP/GraphBLAS; similar ideas follow for the CCS format. For a matrix $\mathbf{A}$, its representation in the CRS format consists of three arrays:

- The `value` array. This stores the actual nonzero elements of $\mathbf{A}$. Then the elements of `value` are of type double. The length of `value` is equal to $nnz\left(\mathbf{A}\right)$
- The `col_indices` array. This stores the column indices of the nonzero elements of $\mathbf{A}$. The array `col_indices` is of type integer and its length is equal to $nnz\left(\mathbf{A}\right)$

- The `ptr_rows` array. This stores the locations or indices in `value` that start a new row and its length is equal to $m + 1$

Based on [5], there are two main goals when dealing with creating data structures for representing and manipulating a sparse matrix on a computer. Memory requirements for storing a sparse matrix should be proportional to its the number of nonzeros, and computing time for sparse matrix computations should be proportional to the operations on the nonzeros of the matrix. The *Sparse Accumulator* (SPA) is an abstract data type that represents a column or a row of a matrix, and its importance lies on the fact that all the computations on the nonzeros of the matrix take place in the SPA, and then the corresponding CRS/CCS arrays of the matrix are updated from its SPA. In fact, it is important to realize that no computations are performed on the sparse matrix directly.

When dealing with sparse matrices, it is quite important to understand that the zero elements are not represented with the value 0 on the CRS/CCS; instead, their locations are not included in the CRS/CCS -these are completely ignored. From this, it follows that nonzeros in a sparse matrix can have the value 0. For example, having $\mathbf{A}(i, j) = 0$ is counted as a nonzero even if its numerical value is zero. We could think of the nonzeros as those elements of the matrix that correspond to an actual numerical value or those positions of the matrix that are occupied.

In ALP/GraphBLAS, a SPA is created from what is presented in [5]. A SPA consists of three arrays: a dense array that store the nonzero values, a dense array of true/false flags, and an unordered array that stores the indices of the occupied elements. Besides, in ALP/GraphBLAS, the SPA is implemented to represent a column, called column-wise SPA and a row of a matrix, called row-wise SPA. The column-major SPA represents a whole row of **A**, which we refer to is the current active row of **A**, and the row-wise SPA represents a whole column of **A**. Henceforth, we refer to and give a few details on the column-major SPA, all of whose arrays are of length equal to the number of columns of **A**. Then, in ALP/GraphBLAS, the SPA of **A** consists of the following arrays.

- The `valBuffer` array. This stores the value of the nonzero elements of the current active row of **A**. Then the type of this array is double
- A boolean array called the `assigned` array. The element `assigned[k]` is true if the kth element of the current active row of **A** is an nonzero - the element `assigned[k]` is said to be assigned; it is false otherwise
- The `stack` array. This keeps in an unordered fashion the indices of the nonzeros of the current active row of **A**. It is of unsigned integer type

In ALP/GraphBLAS, the SPA of any matrix is accessed to and modified by the `Coordinates` class declared in the `coordinates.hpp` file. For any matrix, its SPA is referred to as its coordinates. A `coordinates` object has two important member variables: `n` and `cap`, which are responsible for storing the current number of nonzeros in the SPA, and for storing the size of the three vectors forming the SPA, respectively. Besides, the coordinates of a matrix possesses three important methods. `coordinates.clear()` sets all the elements of `assigned` to false and `n`=0. The method `coordinates.assign(j)` checks if the `assigned[j]` is assigned; if it is not, then it assigns `assigned[j]` to true, increases `n` by one, and writes the index j into `stack` so that `stack[n] = j`. Similarly, the method `coordinates.index(k)` retrieves `stack[k]`.

Therefore, when a new matrix is created in ALP/GraphBLAS, nine arrays are allocated on memory, six for the CRS and CCS formats, and three for the SPA. In comparison to the CRS and CCS formats, we notice that the sizes of each array of the SPA remain unchanged during the whole execution of a program; the length of each of the arrays forming the SPA is fixed since it does not depend on $nnz(\mathbf{A})$. The SPA data structure of a matrix is used for level-3 operations only and there is a single SPA data structure for each matrix in ALP/GraphBLAS.

5.2 Reference backend

In the current implementation of the reference backend ALP/GraphBLAS, the primitive `grb::mxm(C, A, B, ring, phase)` is responsible for computing the $\mathbf{C} = \mathbf{AB}$. The product operator and its identity element, together with the addition operator and its identity element are read from parameter `ring`. The parameter `phase` could take two values only: `RESIZE` or `EXECUTE`; the latter is the default value, and calling `grb::mxm(C, A, B, ring, EXECUTE)` is equivalent to calling `grb::mxm(C, A, B, ring)`. From this point forward, the spM-spM multiplication implemented in ALP/GraphBLAS is referred to as `grb::mxm`.

Code 5.1 depicts how the primitive `grb::mxm(C, A, B, ring, phase)` is implemented in the reference backend; this primitive is found in the `blas3.hpp` file. This code is a simplification that highlights the principal ideas of that primitive. Importantly, this primitive together with all level-3 primitives that output a matrix are implemented as out-of-place operations. That is, the existing contents of $\mathbf{C}$ previous to the execution of `grb::mxm` are always ignored.

```
    RC mxm(C, A, B, ring, phase)
    {
        m = numberRows( C );
        n = numberCols( C );

        // counter that stores nonzeros of C
        nonzeros_C = 0;
        CRS_A = getCRS(A);
        CRS_B = getCRS(B);
        CRS_C = getCRS(C);

        coordinatesC = getCoordinates(C);

        // RESIZE PHASE
        // count number of nonzeros in C
        if( phase == RESIZE )
        {
            // traverse one row of C at a time
            for( i = 0; i < m; ++i )
            {
                // for each row of C, we set coordinatesC to its default state
                coordinatesC.clear();
                // end_row_A - start_row_A is the number of nonzeros in A(i, :)
                start_row_A= CRS_A.ptr_rows[i];
                end_row_A = CRS_A.ptr_rows[i+1];

                // traverse all nonzeros of row i of A
                for( k = start_row_A; k < end_row_A; ++k)
                {
                    // get column index of current nonzero of A
                    k_col = CRS_A.col_indices[k];

                    // k_col also indicates the row of B to use
                    // end_row_B - start_row_B is the number of nonzeros in B(
                        k_col, :)
                    start_row_B = CRS_B.ptr_rows[k_col];
                    end_row_B = CRS_B.ptr_rows[k_col+1];

                    // traverse all nonzeros of B(k_col, :)
                    for( l = start_row_B; l < end_row_B; ++l)
                    {
                        // get column index of current nonzero of B
                        l_col = CRS_B.col_indices[l];
                        // check if assigned[l_col] is assigned;
                        if( !coordinatesC.assign(l_col))
                        {
                            // increase counter of nonzeros
                            ++ nonzeros_C;
                        }
                    }
                }
            }

            // at this point, nonzeros_c stores the total nnz(C)
            // resize of matrix C
            resize( C, nonzeros_c );
        } // END of RESIZE phase

        // EXECUTE PHASE
        if( phase == EXECUTE )
        {
            // counter that stores nonzeros of C
            nonzeros_C = 0;

            // traverse one row of C at a time
            for( i = 0; i < m; ++i )
            {
                // for each row of C, we set coordinatesC to its default state
                coordinatesC.clear();
                // end_row_A - start_row_A is the number of nonzeros in A(i, :)
                start_row_A= CRS_A.ptr_rows[i];
                end_row_A = CRS_A.ptr_rows[i+1];

                // traverse all nonzeros of row i of A
                for( k = start_row_A; k < end_row_A; ++k)
                {
                    // get column index of current nonzero of A
                    k_col = CRS_A.col_indices[k];

                    // k_col also indicates the row of B to use
                    // end_row_B - start_row_B is the number of nonzeros in B(
                        k_col, :)
                    start_row_B = CRS_B.ptr_rows[k_col];
                    end_row_B = CRS_B.ptr_rows[k_col+1];

                    // traverse all nonzeros of B(k_col, :)
                    for( l = start_row_B; l < end_row_B; ++l)
                    {
                        // get column index of current nonzero of B
                        l_col = CRS_B.col_indices[l];
                        // check if assigned[l_col] is assigned;
                        if( !coordinatesC.assign(l_col))
                        {
                            // accumulate the value CRS_A.value[k]*CRS_B.value[l]
                                into coordinatesC.valBuffer[l_col]
                            coordinatesC.valBuffer[l_col] += CRS_A.value[k]*CRS_B.
                                value[l];
                        }
                    }
                }
                // update of the CRS format of C
```

```
98                 // n is the number of elements of assigned that are true
99                 for(k = 0; k < n; ++k)
100                {
101                    // retrieve column index of nonzero element k
102                    j = coordinatesC.index(k);
103                    // update element nonzeros_C of CRS_C.col_indices
104                    CRS_C.col_indices[nonzeros_C] = j;
105                    // update element nonzeros_C of CRS_C.value
106                    CRS_C.value[nonzeros_C] = coordinatesC.valBuffer[j];
107                    // increase number of nonzeros
108                    ++nonzeros_C;
109                }
110                // update CRS_C.ptr_rows
111                CRS_C.ptr_rows[i+1] = nonzeros_C;
112            }
113        }
114        return SUCCESS;
```
Code 5.1: `grb::mxm` implementation in the reference backend

When $\mathbf{C}$ is initialized, $cap(\mathbf{C}) = nnz(\mathbf{C}) = \max(\#rows, \#columns)$. Due to the fact that the sparsity of $\mathbf{C}$ is unknown before the actual computation of $\mathbf{C} = \mathbf{AB}$, there is no guarantee that $\mathbf{C}$ at initialization is able to store all its nonzeros. Then there is a need for counting $nnz(\mathbf{C})$ before computing $\mathbf{C} = \mathbf{AB}$. This is what we refer to as the *resize* or *symbolic* phase of `grb::mxm`; this corresponds to calling `grb::mxm(C, A, B, ring, RESIZE)`.

In Code 5.1, the symbolic phase occurs from lines 16 to 56. The for-loop in line 19 means that the counting of nonzeros of $\mathbf{C}$ is performed one row of $\mathbf{C}$ at a time and the variable `nonzeros_C`, line 7, stores the total number of nonzeros of $\mathbf{C}$, which corresponds to $nnz(\mathbf{C})$. One of the most important aspects of the resize phase of `grb::mxm` is that the actual values of matrices $\mathbf{A}$ and $\mathbf{B}$ are ignored; for this phase, the number of occupied positions in $\mathbf{C}$ is counted. For this purpose, it is worth noting how the arrays `ptr_rows` and `col_indices` of the CRS formats of $\mathbf{A}$ and $\mathbf{B}$ are utilized to check where the nonzeros of $\mathbf{A}$ are. Once `nonzeros_C` is computed, the ALP/GraphBLAS primitive `resize(C, nonzeros_C)` in line 55 is called and this is responsible for allocating sufficient memory for the arrays `values` and `col_indices` of the CRS of $\mathbf{C}$ but modifies none of the elements of these arrays. If there is no memory available, `grb::mxm(C, A, B, ring, RESIZE)` returns with an error code.

After calling `grb::mxm(C, A, B, ring, RESIZE)`, and assuming there is sufficient memory to store $nnz(\mathbf{C})$, it is guaranteed that $\mathbf{C}$ can hold all its nonzeros, and the elements of the three arrays of the CRS of $\mathbf{C}$ are not changed. The actual computation of $\mathbf{C} = \mathbf{AB}$ happens when invoking `grb::mxm(C, A, B, ring, EXECUTE)`, which is calle the computational phase of `grb::mxm`. This phase is performed from lines 56 to 112 in Code 5.1.

During the computational phase, the elements of the three arrays forming the CRS of $\mathbf{C}$ are modified so that they store the value and positions of the nonzeros of $\mathbf{C}$. Avoiding calling `grb::mxm(C, A, B, ring, RESIZE)` before `grb::mxm(C, A, B, ring, EXECUTE)` does not guarantee correctness of the computation of $\mathbf{C} = \mathbf{AB}$ unless there is certainty that the initial $cap(\mathbf{C})$ suffices.

In addition, let assume us that for a program we are interested in computing $\mathbf{C} = \mathbf{AB}$ followed by $\mathbf{D} = \mathbf{CA}$, where $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$ are given matrices. In the reference backend, invoking `grb::mxm(C, A, B, ring, RESIZE)` immediately followed by `grb::mxm(D, C, A, ring, RESIZE)` does not lead to correct results since the second primitive requires the elements of the arrays `col_indices` and `ptr_rows` of the CRS of $\mathbf{C}$ to be updated depending on the sparsity pattern of $\mathbf{C}$. Then, the order of operations that guarantee correct results as long as sufficient memory is available must be `grb::mxm(C, A, B, ring, RESIZE)`, `grb::mxm(C, A, B, ring)`, `grb::mxm(D, C, A, ring, RESIZE)` and `grb::mxm(D, C, A, ring)`.

Let us take an example to view how the CRS formats and SPA of a matrix are used to compute `grb::mxm`. We perform the matrix product of $\mathbf{C} = \mathbf{AB}$ for matrices given in Figures 5.1 and 5.2, respectively. The idea of displaying matrices as shown in the following Figures in this section is taken from [10].

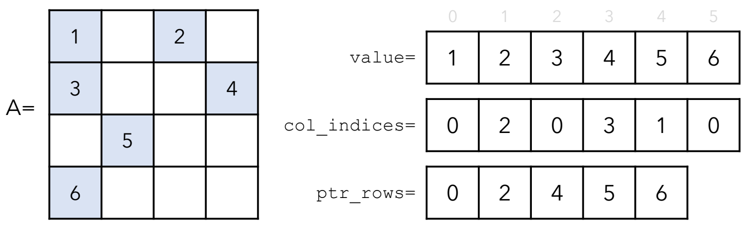


Figure 5.1: **A** and its CRS format.

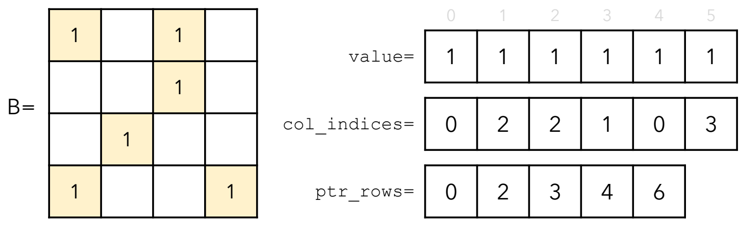


Figure 5.2: **B** and its CRS format.

After the execution of `grb::mxm(C, A, B, ring, RESIZE)`, the CRS and SPA of **C** looks like in Figure 5.3. In the same figure, `F` in `assigned[i]` stands for false; `T` stands for true.

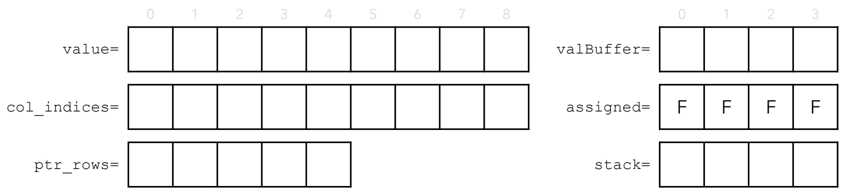


Figure 5.3: CRS (left) and SPA (right) of **C** after `grb::mxm(C, A, B, ring, RESIZE)`.

Then, `grb::mxm(C, A, B, ring, EXECUTE)` is invoked. Following lines 59 to 112 in Code 5.1, the first row to be computed is $\mathbf{C}(0,:)$, i.e., the current active row is $\mathbf{C}(0,:)$. According to the row-wise Gustavson algorithm, to compute $\mathbf{C}(i,:)$ we need $\mathbf{A}(i,:)$ and $\mathbf{B}$ as show in (2.1), [10].

$$\mathbf{C}(i,:) = \sum_k \mathbf{A}(i,k)\mathbf{B}(k,:) \tag{2.1}$$

To compute $\mathbf{C}(0,:)$, we notice that $\mathbf{A}(0,0) = 1$, which means that `k_col = 0` in line 77. This means that the nonzeros in row $\mathbf{B}(0,:)$ are needed, thus we traverse that row in the for-loop of lines 85 to 95. We observe that the difference `end_row_B - start_row_B`, lines 81 and 82, is equal to the total nonzeros in row $\mathbf{B}(0,:)$. The first nonzero in $\mathbf{B}(0,:)$ has a column index `l_col=0`, line 88, then `coordinatesC.assign()` in line 90 is responsible for doing the following: `assigned[l_col] = T`, `stack[0] = l_col`, and `valBuffer[l_col] =` $\mathbf{A}(0,0) * \mathbf{B}(0,0) = 1$. Likewise, the second nonzero in $\mathbf{B}(0,:)$ has a column index `l_col=2`, line 88, then `coordinatesC.assign()` does the following: `assigned[l_col] = T`, `stack[1] = l_col`, and `valBuffer[l_col] =` $\mathbf{A}(0,0) * \mathbf{B}(0,2) = 2$. No more nonzeros are found in $\mathbf{B}(0,:)$, then the current state of the SPA of $\mathbf{C}$ is presented in Figure 5.4. We remark that the CRS of $\mathbf{C}$ is not modified at this point; its state is as shown in Figure 5.3 (left).

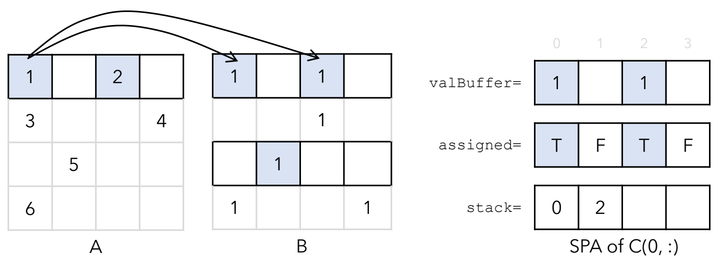


Figure 5.4: Current state of SPA of **C**.

Now, we move to the next iteration of the for-loop in line 74; the next nonzero in $\mathbf{A}(0,:)$ is $\mathbf{A}(0,2)$, which leads to `k_col=2`, then row $\mathbf{B}(2,:)$ is needed. There is a unique nonzero in $\mathbf{B}(2,:)$, which has a column index `l_col=1`, line 88, then, using `coordinatesC.assign(l_col)` in line 90, `assigned[l_col]=T`, `stack[2]=l_col`, and `valBuffer[l_col] =` $\mathbf{A}(0,2) * \mathbf{B}(2,1) = 1$. The current state of the SPA of $\mathbf{C}$ is presented in Figure 5.5.

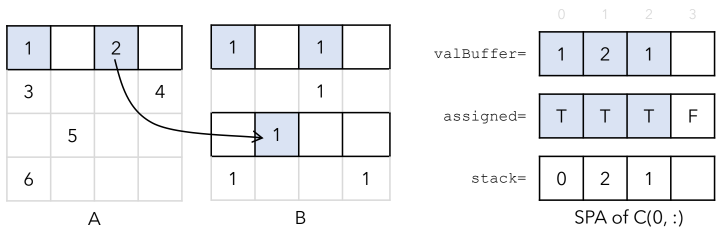


Figure 5.5: Current state of SPA of **C**.

Since there are no more nonzeros in $\mathbf{A}(0,:)$, the SPA of $\mathbf{C}(0,:)$ is such that `valBuffer` stores its nonzero values, `stack` stores the column indices of its nonzeros, and the total number of elements of `assigned` that are true corresponds to the total number of nonzeros in that row.

In order to update the CRS of $\mathbf{C}$ from its SPA, the array `stack` is traversed one element at a time starting from its initial element as follows. A counter that starts with a value of 0 is needed to keep track of how many nonzeros of row $\mathbf{C}(0,:)$ have been updated in the CRS of $\mathbf{C}$; we call this `counter=0`. Then we commence by observing that `stack[counter] = 0`, which implies that `col_indices[counter] = 0` and `value[counter] = valBuffer[0] = 1`. At this point, one nonzero element has been set, then `counter=1`. Thus, the next element in `stack` corresponds to `stack[counter] = 2`, which means that `col_indices[counter] = 2` and `value[counter] = valBuffer[2]=1`, then `counter=2`. The same happens for the last element in `stack`; that is, `stack[2]=1`, which means that `col_indices[counter]=1`, `value[counter]=valBuffer[1]=2`, and `counter=3`. The final value of `counter` indicates how many nonzeros there are in $\mathbf{C}(0,:)$, which is the value written to `ptr_rows[1]`. Figure 5.6 displays the state of the CRS $\mathbf{C}$ after the computation of $\mathbf{C}(0,:)$ has finished. We notice that the SPA of $\mathbf{C}$ is used only to update the arrays `value` and `col_indices` of its CRS; the array `ptr_rows` depends only on the number of nonzeros of $\mathbf{C}$, not where they are.

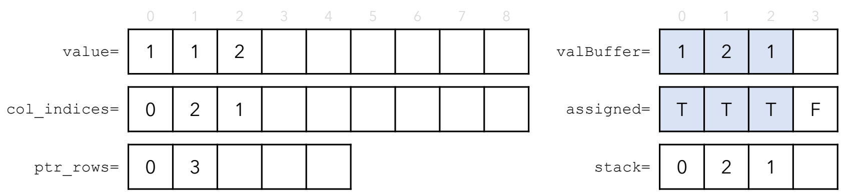


Figure 5.6: CRS (left) and SPA (right) of $\mathbf{C}$ when of $\mathbf{C}(0,:)$ is computed.

After updating the CRS of $\mathbf{C}$ from its SPA, the latter is restarted to its initial state, line 68, as shown in Figure 5.3 (right). The most important point to keep in mind is that the same SPA is utilized for every row of $\mathbf{C}$; there is no one SPA for each row of $\mathbf{C}$. Therefore, the same steps are repeated to compute the remaining rows of $\mathbf{C}$. After computing all rows of $\mathbf{C}$, its CRS is shown in Figure 5.7.

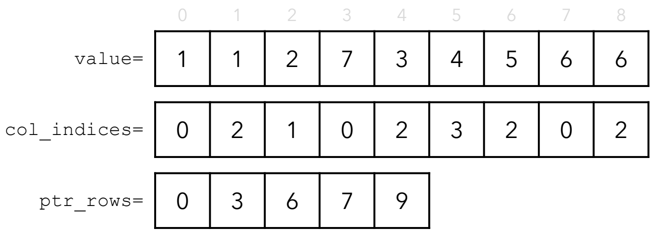


Figure 5.7: CRS of $\mathbf{C}$ when all its rows have been computed.

5.3 Nonblocking backend

The implementation of the `grb::mxm` primitive presented in this work can be found on `https://github.com/Algebraic-Programming/ALP` on branch `nonblocking-backend-implementation-blas3`.

On the other hand, to enable utilizing the same algorithm in Code 5.1 in the nonblocking reference, there are a few changes to be made to the implementation shown in that code.

Firstly, let us we imagine that in a program we have the primitives corresponding to the operations $\mathbf{C} = \mathbf{AB}$ and $\mathbf{x} = \mathbf{Cb}$ for $\mathbf{A}, \mathbf{B}$, and $\mathbf{b}, \mathbf{x}$ known. We observe that both operations utilize matrix $\mathbf{C}$, then from a data dependence analysis perspective, the primitives of these operations share data and both would be added into the same pipeline. Thus, the computation of $\mathbf{C}$ in the nonblocking backend should done such that it allows having both primitives in the same pipeline. Moreover, we recall that vector $\mathbf{x}$ is computed one element at a time. That is, to compute the ith element of $\mathbf{x}$, the whole row $\mathbf{C}(i,:)$ and the entire vector $\mathbf{b}$ are required.

Then, as long as row $\mathbf{C}(i,:)$ has been computed, it is ensured that the ith element of $\mathbf{x}$ can be computed too. We can apply the same reasoning if $\mathbf{C}$ and $\mathbf{x}$ are computed in tiles. For example, let us consider tile $T1 = \mathbf{C}(i:j,:)$, then as soon as such rows of $\mathbf{C}$ are computed, exactly the same rows of $\mathbf{x}$ can be computed no matter what the remaining rows of $\mathbf{C}$ contain.

In addition, let assume that in a program we are interested in computing $\mathbf{C} = \mathbf{AB}$ followed by $\mathbf{D} = \mathbf{CA}$ in the nonblocking mode. Once again, the primitives of these two operations share data and then they would belong to the same pipeline. Similar to the spM-spV product case, computing $\mathbf{D}(i,:)$ requires $\mathbf{C}(i,:)$ and the entire matrix $\mathbf{A}$, thus once $\mathbf{C}(i,:)$ has been computed, the corresponding row of $\mathbf{D}(i,:)$ can be computed. We can apply the same reasoning if we compute $\mathbf{C}$ and $\mathbf{D}$ by tiling row-wise.

Therefore, this leads to stating that for the nonblocking execution `grb::mxm`, the output matrix of that operation should be computed row-wisely; the most important aspect is to realize that each tile of the output matrix performs its computations independently from the other tiles, then each tile of the can be computed safely in parallel.

Let us assume that $\mathbf{C} = \mathbf{AB}$ followed by $\mathbf{D} = \mathbf{CA}$ are to be computed. To achieve the execution of these two operations in the nonblocking backend, we should be able to delay their computational phases. This implies that calling `grb::mxm(C, A, B, ring, RESIZE)` immediately followed by `grb::mxm(D, C, A, ring, RESIZE)` should lead to correct results; this means, that the latter primitive must be aware of where the nonzeros of the former primitive are without enforcing its computational phase, i.e. it is not mandatory to call the primitive `grb::mxm(C, A, B, ring)` before `grb::mxm(D, C, A, ring, RESIZE)` for correct results.

This implies that during `grb::mxm(C, A, B, ring, RESIZE)` in the nonblocking backend, the `col_indices` and `ptr_rows` arrays of the CRS of $\mathbf{C}$ must be first resized -this is all the same phase does in the reference backend-, and then their elements must updated such that they store the sparsity pattern of $\mathbf{C}$, i.e., the positions of its nonzeros. Once this is done, calling `grb::mxm(D, C, A, ring, RESIZE)` will know where the nonzeros of $\mathbf{C}$ are to update the sparsity pattern of $\mathbf{D}$, and this information is stored in the `col_indices` and `ptr_rows` arrays of the CRS of $\mathbf{D}$.

As a result, `grb::mxm(C, A, B, ring, EXECUTE)` and `grb::mxm(D, C, A, ring, EXECUTE)` are responsible for the computational phases that, in this case, means modifying the elements of the `value` array of the CRS of $\mathbf{C}$ and $\mathbf{D}$, respectively, and the execution of these primitives are delayed. This manner of updating the CRS of the output matrices is a key difference with respect to how the `grb::mxm` primitive works in the reference backend, and is the central component that enables keeping the same algorithm in Code 5.1 for the nonblocking execution of `grb::mxm`.

Consequently, in the nonblocking execution mode, `grb::mxm` still uses two phases: symbolic and computational phases. It is quite important to indicate that the symbolic phase of `grb::mxm` is not delayed; that is, it not added into any pipeline. For example, after calling `grb::mxm(C, A, B, ring, RESIZE)`, the elements of `ptr_rows` and `col_indices` of the CRS of $\mathbf{C}$ are updated. Conversely, the execution of `grb::mxm(C, A, B, ring, EXECUTE)` will be deferred by using three lambda functions, about which we now discuss by using an example.

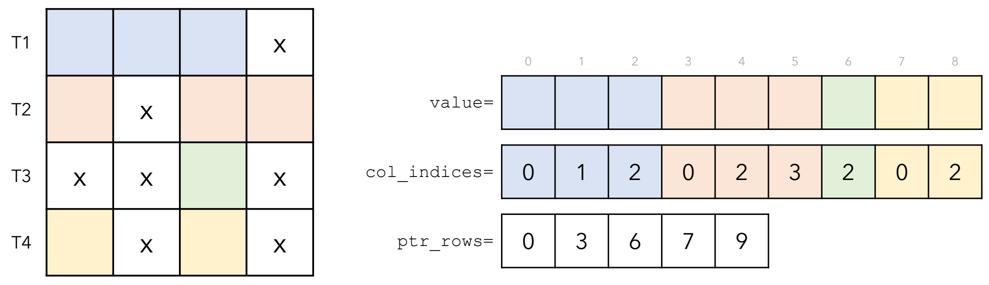
To illustrate how the `grb::mxm` primitive works in the nonblocking mode of ALP/GraphBLAS, let use imagine that the operations $\mathbf{C} = \mathbf{AB}$, $\mathbf{D} = \mathbf{B} + \mathbf{C}$ (element-wise addition), and $s = \sum_{i,j=1}^{3} \mathbf{D}(i,j)$ are to be performed in that order in a program. Matrices $\mathbf{A}, \mathbf{B}$ are given in Figures 5.1 and 5.2, respectively. These operations can be written in ALP/GraphBLAS as shown in Code 5.2; details of initialization of matrices and error checking are omitted.

```
int main( int argc , char ** argv )
{
    // initialization A, B, C, D, E
    // ...

6      // fill matrices A and B with data
7      // ...
8
9      // initizaliton of ring and monoid
10     const Semiring< grb::operators::add< double >, grb::operators::mul< double >,
           grb::identities::zero, grb::identities::one > ring;
11
12     const Monoid< grb::operators::add< double >, grb::identities::zero > monoid;
13
14     // C = AB
15     grb::mxm( C, A, B, ring, RESIZE );
16     grb::mxm( C, A, B, ring );
17
18     // D = B + C
19     grb::eWiseApply( D, B, C, monoid, RESIZE );
20     grb::eWiseApply( D, B, C, monoid );
21
22     // Computation of s
23     double s = 0;
24     grb::foldl( s, D, monoid );
25
26     // ...
27 }
```

Code 5.2: Code example of level-3 operations in nonblocking execution mode.

We are using the same terminology of and ideas detailed in section 4.3.3 to describe how these operations are added into `pipelines` and below we refer to the lines in Code 5.2. All primitives called in 5.2 were implemented in the nonblocking backend for this work.

Let us detail how $\mathbf{C} = \mathbf{AB}$ is computed in the nonblocking backend. The resize of `values` and `col_indices` of $\mathbf{C}$ happens exactly as in the reference backend, lines 16 to 56 in Code 5.1. After this task, these arrays are updated in lines 31 to 52 of Code 5.3. We observe that these lines correspond to the lines of the computational phase in Code 5.1 with the major difference that `value` of $\mathbf{C}$ is not modified. After `grb::mxm(C, A, B, ring, RESIZE)` has returned, `ptr_rows` and `col_indices` of C contain the positions of the nonzeros and the total nnz of $\mathbf{C}$ is known at this point, see Figure 5.8. It has been explained how the computational phase of `grb::mxm` occurs in the reference backend, then we omit details herein. The resize phase is not computed in parallel.



(a) Sparsity pattern of **C**.   (b) Arrays of the CRS of **C**.

Figure 5.8: State of **C** after the symbolic phase of `grb::mxm`, nonblocking backend.

Then, when invoking `grb::mxm(C, A, B, ring)`, line 16, firstly, that primitive is added as the first stage of Pip1.

The first lambda function of the computational phase is defined in line 60 and it is responsible for counting the number of nonzeros in each tile of **C** and is called `count_nonzeros_local`. This function allows each tile to compute independently its own number of nonzeros and write this value into an array called `nnz_tiles`. This lambda function takes the lower and upper bounds of each tile as parameters. The main task of this lambda is done by the same algorithm used to count $nnz(\mathbf{C})$ during the resize phase of `grb::mxm` in the reference backend, lines 21 to 50 in Code 5.1.

However, there is an important modification to make concerning the SPA of **C**. When Pip1 is executed, each tile of **C** works independently from the others, thus they cannot use the SPA of **C**; otherwise, all tiles will be accessing to and modifying the same memory location at the same time; this causes data races. Then, for each matrix of the nonblocking backend, an array of SPA is created and this contains as many SPA's as available threads on the system. Inside the `count_nonzeros_local` function, each tile manipulates its own SPA. Moreover, each tile has a unique identifier called `tile_id` that we can deduce from its lower and upper bounds.

Similarly, the second lambda function, called `func_prefix_sum`, is in charge of computing the accumulated sum of `nnz_tiles`. We refer to the array `value` of the SPA of **C**. In our example in Figure 5.8, even thought having `nnz_tiles` suffices for updating `value` in parallel when Pip1 is executed, the existence of this lambda is justified by noticing that tile T1 is responsible for updating `value[0]`, `value[1]` and `value[2]`, tile T2 for `value[3]`, `value[4]` and `value[5]`, tile T3 for `value[6]`, and tile T4 for `value[7]` and `value[8]`. This highlights the fact that the initial position at which each tile starts writing at `value` is distinct. Thus, each tile is responsible for computing an accumulated sum from `nnz_tiles` to know where to start modifying `value`. To avoid each tile performing this sum, we compute the accumulated sum once for all tiles; this result is stored in `prefix_sum_nnz_tiles`. From this, we observe that each tile can access its position of `value` at which to write by using `prefix_sum_nnz_tiles[tile_id]` without having to perform any accumulated sum.

Then, a third lambda function, called `func`, is defined and it is responsible for the actual computational phase of `grb::mxm`; it modifies only the elements of `value` of the CRS format of **C** based on the values on the inputs **A** and **B**. As we observe in lines 87 to 112, the tasks `func` performs are essentially the same algorithm of the computational phase of `grb::mxm` in Code 5.1. `func` takes as parameters the pipeline where the primitive corresponding to $\mathbf{C} = \mathbf{AB}$ is added, and the lower and upper bounds of each tile. None of the previous three lambdas is executed after being defined; these are passed to a pipeline, line 115.

The principal idea of the `grb::mxm` primitive in the nonblocking backend is to enable `func` to use the same algorithm of `grb::mxm` in the reference backend, Code 5.1. This is the reason for which we need the functions `count_nonzeros_local` and `func_prefix_sum` before the computational phase. We can think of these lambda functions are preparatory steps that are needed to compute each tile of **C** in parallel. Moreover, we need to have two separated lambda functions since it is not possible to compute the prefix sum unless `nnz_tiles` is fully updated. `func_prefix_sum` requires vector `nnz_tiles` to be fully updated with the total number of nonzeros in each tile of **C**; computing the prefix sum of `nnz_tiles` at the same time such an array is filled in could lead to incorrect results due to parallel execution.

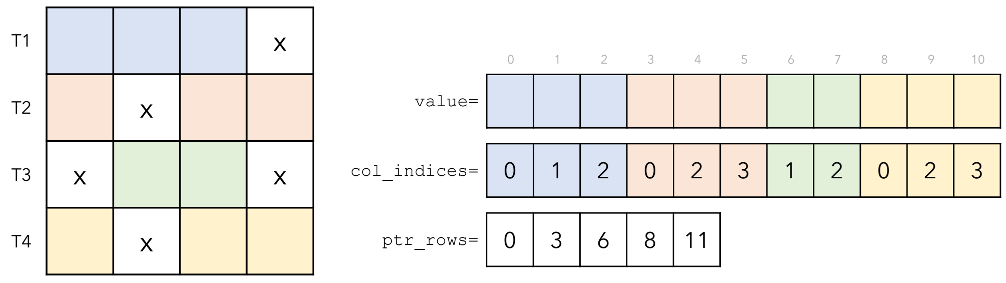
Then, the implementation of the computational phase of `grb::mxm` in the nonblocking execution mode has the form presented in Code 5.3.

```

    RC mxm( C, A, B, ring, phase)
    {
        m = numberRows( C );
        n = numberCols( C );

        // counter that stores nonzeros of C
        nonzeros_C = 0;
        CRS_A = getCRS(A);
        CRS_B = getCRS(B);
        CRS_C = getCRS(C);

        coordinatesC = getCoordinates(C);

        // RESIZE PHASE
        if( phase == RESIZE )
        {
            // Count the total number of nonzeros of C; the same procedure shown
                from lines 19 to 51 of Code 6.1 is followed
            for( i = 0; i < m; ++i )
            {
                // ...
            }

            // at this point, nonzeros_C stores the total nnz(C)
            // resize of matrix C
            resize( C, nonzeros_C );

            // From this point, the modifications for the nonblocking backend
                start

            // reset nonzeros_C to zero
            nonzeros_C = 0;

            // traverse one row of C at a time
            for( i = 0; i < m; ++i )
            {
                // same algorithm shown in lines 21 to 50 in Code 6.1
                // ...

                // update two arrays of the CRS format of C
                for(k = 0; k < _n; ++k)
                {
                    // retrieve column index of nonzero element k
                    j = coordinatesC.index(k);
                    // update element nonzeros_C of CRS_C.col_indices
                    CRS_C.col_indices[nonzeros_C] = j;
                    // increase number of nonzeros
                    ++nonzeros_C;
                }
                // update CRS_C.ptr_rows
                CRS_C.ptr_rows[i+1] = nonzeros_C;
            }
            // CRS_C.ptr_rows and CRS_C.col_indices contain information about the
                location of the nonzeros of C
        }

        // EXECUTE PHASE
        if( phase == EXECUTE )
        {
            // lambda  function to count the nnz in each tile
            func_count_nonzeros = [ ... ] (lower_bound, upper_bound)
            {
                // local counter of nonzeros for tile whose bounds are lower_bound
                    and upper_bound
                nnz_current_tile = 0;

                // get tile_id
                tile_id = getTileID(lower_bound, upper_bound );

                for( i = lower_bound; i < upper_bound; ++i )
                {
                    // count number of nonzeros in tile whose bounds are lower_bound and upper_bound
                    // same algorithm shown in lines 21 to 50 in Code 6.1
                    // ...
                }

                // assign corresponding element nnz_tiles[tile_id]
                nnz_tiles[tile_id] = nnz_current_tile;
            };

            // lambda function to compute the prefix sum of nnz_tiles
            func_prefix_sum = [ ... ]()
            {
                // prefix sum of nnz_tiles is computed
                // ...
            };

            // lambda function that corresponds to the actual computational phase
            func = [...](pipeline, lower_bound, upper_bound)
            {
                for( i = lower_bound; i < upper_bound; ++i )
                {
                    // get tile_id
                    tile_id = getTileID(lower_bound, upper_bound );

                    // read prefix sum for tile_id
                    nonzeros_C_local = getPrefixSum(C, tile_id);

                    // compute C = A*B for tile whose bounds are lower_bound, upper_bound. This modifies the array value of the CRS of C
                    // same algorithm in lines 67 to 96 in Code 6.1
                    // ...

                    // modifications for nonblocking backend
                    // update one array of the CRS format of C
                    for(k = 0; k < n; ++k)
                    {
                        // retrieve column index of nonzero element k
                        j = coordinatesC.index(k);
                        // update element nonzeros_C of CRS_C.value
                        CRS_C.value[nonzeros_C_local] = coordinatesC.valBuffer[j];
                        // increase number of nonzeros
                        ++nonzeros_C_local;
                    }
            };

            // none of the lamnda function is executed after being defined, they are added into a pipeline
            ret = ret ? ret : internal::le.addStageLevel3(
                    std::move(func),
                    // name of operation
                    internal::Opcode::BLAS3_MXM_GENERIC,
                    // size of output matrix
                    grb::nrows(C),
                    // size of data type in matrix C
                    sizeof(OutputType),
                    // dense_descr
                    true,
124                     // dense_mask
125                     true,
126                     // matrices for mxm
127                     &A, &B, &C,
128                     std::move(func_count_nonzeros), std::move(func_prefix_sum) );
129 
130             return ret;
131         }
```

Code 5.3: `grb::mxm` implementation in the nonblocking backend.

We refer to lines in Code 5.3. In lines 71, 89 and 97 the three aforementioned lambda functions are created. One key point to keep in mind is that the lambdas `count_nonzeros_local` and `func_prefix_sum` do not take a `pipeline` as a parameter and `func` does take a pipeline as a parameter. That is, for the computational phase, which occurs in `func`, two extra lambda functions are added into the pipeline `func` belongs to. This `pipeline` parameter is used during the data dependence analysis done in the `LazyEveluation` class. In line 103, the three lambda functions corresponding to `grb::mxm` in the `EXECUTE` mode are added into the pipeline by using the function `le.addStageLevel3`, which is meant to adding level-3 operations into a pipeline. Creating the function `le.addStageLevel3` to the `LazyEvaluation` class is an implementation for this project.

Afterwards, in line 19 of Code 5.2, the primitive `grb::eWiseApply(D,B,C,monoid,RESIZE)` is found. In the reference backend this primitive consists of a resize and a computational phase as `grb::mxm`. To implement that primitive in the nonblocking backend, we follow analogous ideas to the ones previously highlighted for `grb::mxm` for both of its phases. Based on the data dependence analysis, `grb::eWiseApply(D,B,C,monoid)` in line 20 of Code 5.2 shares data with the first stage of Pip1 and then it is added as its second stage; this primitive does not lead to Pip1's execution. After calling `grb::eWiseApply(D,B,C,monoid,RESIZE)` and `grb::eWiseApply(D,B,C,monoid)`, Pip1 contains two stages and the CRS of **D** is displayed in 5.9.



(a) Sparsity pattern of **D**. (b) Arrays of the CRS of **D**.

Figure 5.9: State of **D** after the symbolic phase of `grb::eWiseApply`.

In line 24, the primitive `grb::foldl(s, D, monoid)` is encountered. There is no resize phase for this primitive since it does not change the sparsity patter of **D**. Despite the fact that we still use three lambda functions for its execution as before for `grb::mxm`, its lambda functions `count_nonzeros_local` and `func_prefix_sum` do not perform any computations. Its function `func` is in charge of the main computation. In addition, since this primitive shares data with the second stage of Pip1, this is added as Pip1's third step. This primitive outputs a scalar; thus after having been added as the last stage of Pip1, the execution of Pip1 starts.

The code snippet shown in 5.4 to demonstrate the order execution of the lambda functions in Pip1. This is a simplification of the `Pipeline::execution()` function, which is declared in the `Pipeline` class, found in the `pipeline.hpp` file. In Code 5.4, it is shown the lines of code of the function responsible for executing a pipeline where the order execution of each of the lambda functions. We include Code 5.4 to emphasize the fact that `count_nonzeros_local` and `func_prefix_sum` are preliminary steps that enable the correct execution of `func`.

```
1  grb::RC Pipeline::execution()
2  {
3      // ...
4
5      // Execution of the lambdas responsible for counting nonzeros in each tile
6      #pragma omp parallel for schedule( dynamic ) num_threads( nthreads )
7          for( tile_id = 0; tile_id < num_tiles; ++tile_id) {
8
9              // compute the lower and upper bounds of tiles
10             lower_bound[tile_id] = getLowerBound(tile_id);
11             upper_bound[tile_id] = getUpperBound(tile_id);
12
13             // all lambdas responsible for counting the number of nonzeros in each
                   tile are computed here
14             for( pt = pbegin_count_mxm(); pt != pend_count_mxm(); ++pt )
15             {
16                 // computation of lambda functions
17                 ( *pt )(lower_bound[tile_id], upper_bound[tile_id]);
18             }
19         }
20
21     #pragma omp barrier
22
23     // Execution of the lambdas responsible for prefix sum
24     for( pt = pbegin_prefix_sum_mxm(); pt != pend_prefix_sum_mxm(); ++pt )
25     {
26         ( *pt )();
27     }
28
29     // ...
30
31     // Execution of the lambdas responsible for computational phase
32     #pragma omp parallel for schedule( dynamic ) num_threads( nthreads )
33         for(tile_id = 0; tile_id < num_tiles; ++tile_id) {
34
35             // compute the lower and upper bounds of each tile
36             lower_bound[tile_id] = getLowerBound(tile_id);
37             upper_bound[tile_id] = getUpperBound(tile_id);
38
39             for(pt = pbegin(); pt != pend(); ++pt)
40             {
41                 ( *pt )(*this, lower_bound[tile_id], upper_bound[tile_id]);
42             }
43             // ...
44         }
45     // ...
46 }
```

Code 5.4: Order execution of lambdas in Pipeline::execution().

From this point forward, we refer to lines in code 5.4. When the execution of Pip1 starts, all the lambda functions called `count_nonzeros_local` are first executed. This is what happens from lines 6 to 19. We observe that the for-loop in line 7 iterates over each tile while the for-loop shown in line 14 iterates through, and executes, all the lambda functions `count_nonzeros_local`.

Then from lines lines 24 to 27, all the functions `func_prefix_sum` are executed; observe that no information about the bounds of the tiles is passed to any of these lambdas, line 26, and that this for-loop does not iterates over the tiles either. In lines 32 to 44, the lambda functions in charge of the computational phases of `grb::mxm`, `grb::eWiseApply` and `grb::foldl` are executed.

On the other hand, concerning level-3 primitives that might change the sparsity pattern of their output matrices like `grb::mxm` and `grb::eWiseApply`, until this point, we have assumed that each of them is first called in the `RESIZE` mode. However, a resize phase may be unnecessary if it is certain that the output matrices have sufficient capacity when they are created. The current implementation of the nonblocking execution mode assumes that all level-3 operations that might change the sparsity pattern of their output matrices are invoked first in the `RESIZE` mode.

5.3.1 Role of coordinates for nonblocking execution for the spMspM product

It is important to make a comment on the reason for which a local coordinates mechanism as the one shown in section 5.3.1 for operations on sparse vectors is not needed when dealing with `grb::mxm` in the nonblocking backend. In the discussion below, we focus on the CRS format only.

In vectors, the local coordinates mechanism briefly described in section 4.3.4 is utilized for correctly manipulating the sparsity pattern of a sparse vector in the nonblocking backend. The major reason being that data races could be found when changing simultaneously the array `stack` of the global view of and the number of nonzeros of that vector.

On the other hand, matrices are stored in the CRS format since they are assumed to be always sparse. That is, changing the number of nonzeros in a matrix impacts directly the memory requirement to store that matrix. However, since a resize phase of `grb::mxm` is assumed to happen before any computation is performed, all memory allocations to store the output matrix of `grb::mxm` happen once during the execution of that primitive, and together with the fact that the output matrix of `grb::mxm` is computed in parallel by tiling row-wise, there are no data races when each tile of the output matrix writes to its own part of the array `value` of the CRS format of that matrix. As a result, the local coordinates mechanism for sparse vectors is not needed for `grb::mxm`. This simplifies significantly implementing the `grb::mxm` primitive in the nonblocking backend by using the spM-spM product algorithm from the reference backend. The same reasoning applies to the primitive `grb::eWiseApply` in the nonblocking backend.

5.4 Masking output of the spMspM product

Let us assume that we are interested in computing the product $\mathbf{C} = \mathbf{AB}$ but we are interested in the nonzeros on the diagonal of $\mathbf{C}$ only. The first approach to achieve this is to compute directly that product and then read those elements of interest. However, by doing this, computational resources would be wasted.

Therefore, the idea of *masking* the output matrix $\mathbf{C}$ comes into play. The main idea behind masking is to compute only those elements of interest. Masking the output of `grb::mxm` consists of selecting which elements $\mathbf{C}$ to compute based on a filter or a mask. This mask is information the user must provide to the program.

Currently, the primitive of `grb::mxm` does not consider any mask on the ouput matrix in any of the backends available for ALP/GraphBLAS. Then, for this project, a masked `grb::mxm` primitive, which we refer to as `grb::masked_mxm`, for the reference and nonblocking backends are implemented while using the same algorithm of `grb::mxm` in the reference backend, see Code 5.1. We can think of this implementation as a further extension for the `grb::mxm` primitive. As in the reference backend, `grb::masked_mxm` consists of a resize and computational phase.

Let us denote the mask matrix for $\mathbf{C}$ by $\underline{\mathbf{C}}$. In the implementation presented herein, $\underline{\mathbf{C}}$ is an object created from the `Matrix` class, which means we can access to its CRS and CCS formats, its SPA, and we can use the available methods of the `Matrix` class on it. The numerical values of $\underline{\mathbf{C}}$ are not important in this context and then, as long as a valid `C++11` data type is used, a matrix of any type can be used as a mask. Indeed, to know what elements of $\underline{\mathbf{C}}$ are occupied, we exclusively use `col_indices` and `ptr_rows` of its CRS format. Matrix $\underline{\mathbf{C}}$ must have the same number of columns and rows as $\mathbf{C}$.

In general, the implementation of $\underline{\mathbf{C}}$ proposed in this work consists of enabling the `grb::mxm` primitive in the reference backend to ingest information on $\underline{\mathbf{C}}$.

The core points of the implementation of `grb::masked_mxm` in the reference backend are shown in Code 5.5.

```
RC mxm_masked(C, C_mask, A, B, ring, phase)
{
    // same lines 3 to 12 in Code 6.1
    // ...

    //get SPA of C_mask
    coordinatesC_mask = getCoordinates(C_mask);

    // RESIZE PHASE
    // count number of nonzeros in C
    if( phase == RESIZE )
    {
        // count number of nonzeros in C_mask.
        // we know that the total number of nonzeros in C is the number of
        // nonzeros in the mask, so we just traverse C_mask
        for( i = 0; i < m; ++i )
        {
            coordinatesC_mask.clear();
            // end_row_C_mask - start_row_C_mask is the number of nonzeros in
                C_mask(i, :)
            start_row_C_mask= CRS_C_mask.ptr_rows[i];
            end_row_C_mask = CRS_C_mask.ptr_rows[i+1];
            // column index of nonzero
            k_col = CRS_C_mask.col_indices[ k ];
            if( !coordinatesC_mask.assign( k_col ) )
            {
                // increase number of nonzeros
                ++nonzeros_C;
            }
        }

        // at this point, nonzeros_C stores the total nnz(C)
        // resize of matrix C
        resize( C, nonzeros_C );

    } // END of RESIZE phase

    // EXECUTE PHASE
    if( phase == EXECUTE )
    {
        // counter that stores nonzeros of C
        nonzeros_C = 0;

        // traverse one row of C at a time
```

```
            for( i = 0; i < m; ++i )
            {
                // we traverse C_mask(i, :) to find the column indices of its nonzeros
                coordinatesC_mask.clear();

                // end_row_C_mask - start_row_C_mask is the number of nonzeros in C_mask(i, :)
                start_row_C_mask= CRS_C_mask.ptr_rows[i];
                end_row_C_mask = CRS_C_mask.ptr_rows[i+1];
                for( k = start_row_C_mask; k < end_row_C_mask; ++k)
                {
                    // column index of nonzero
                    k_col = CRS_C_mask.col_indices[ k ];
                    coordinatesC_mask.assign( k_col );
                }

                // at this point, the stack array of coordinatesC_mask has all the column indices of the nonzeros in C_mask(i, :)
                // then we sort the stack array in ascending order
                sort(coordinatesC_mask.stack);

                // for each row of C, we set coordinatesC to its default state
                coordinatesC.clear();
                // end_row_A - start_row_A is the number of nonzeros in A(i, :)
                start_row_A= CRS_A.ptr_rows[i];
                end_row_A = CRS_A.ptr_rows[i+1];

                // traverse all nonzeros of row i of A
                for( k = start_row_A; k < end_row_A; ++k)
                {
                    // get column index of current nonzero of A
                    k_col = CRS_A.col_indices[k];

                    // k_col also indicates the row of B to use
                    // end_row_B - start_row_B is the number of nonzeros in B(k_col, :)
                    start_row_B = CRS_B.ptr_rows[k_col];
                    end_row_B = CRS_B.ptr_rows[k_col+1];

                    // traverse all nonzeros of B(k_col, :)
                    for( l = start_row_B; l < end_row_B; ++l)
                    {
                        // get column index of current nonzero of B
                        l_col = CRS_B.col_indices[l];

                        // check if l_col is stack of C_mask(i, :)
                        find = binary_search(l_col, stack);

                        if( find )
                        {
                            // check if assigned[l_col] is assigned;
                            if( !coordinatesC.assign(l_col))
                            {
                                // accumulate the value CRS_A.value[k]*CRS_B.value[l] into coordinatesC.valBuffer[l_col]
                                coordinatesC.valBuffer[l_col] += CRS_A.value[k]*CRS_B.value[l];
                            }
                        }
                    }
                }
```

```
100                 // update of the CRS format of C
101                 for(k = 0; k < n; ++k)
102                 {
103                     // retrieve column index of nonzero element k
104                     j = coordinatesC.index(k);
105                     // update element nonzeros_C of CRS_C.col_indices
106                     CRS_C.col_indices[nonzeros_C] = j;
107                     // update element nonzeros_C of CRS_C.value
108                     CRS_C.value[nonzeros_C] = coordinatesC.valBuffer[j];
109                     // increase number of nonzeros
110                     ++nonzeros_C;
111                 }
112                 // update CRS_C.ptr_rows
113                 CRS_C.ptr_rows[i+1] = nonzeros_C;
114             }
115         } // END of EXECUTE phase
116 
117         return SUCCESS;
118     }
```

Code 5.5: `grb::mxm_masked` implementation in the reference backend

From this point, we refer to lines in Code 5.5. During the resize phase, lines 18 to 42, we observe that we do not use any of the input matrices to count $nnz(\mathbf{C})$; instead, we use $\underline{\mathbf{C}}$ to determine how many nonzeros there are in $\mathbf{C}$. We exclusively traverse $\underline{\mathbf{C}}$ row by row, and for each row we count its number of nonzeros. This is the reason for which in lines 23 to 36, no information about $\mathbf{A}$ or $\mathbf{B}$ is used.

The computational phase happens in lines 45 to 71. Let us say that row $\mathbf{C}(i,:)$ is being computed. The purpose of lines 54 to 64 is to store the column indices of the nonzeros in $\underline{\mathbf{C}}(i,:)$. The array `stack` of the CRS of $\underline{\mathbf{C}}$ stores such indices. Then, the elements of `stack` are sorted in ascending order. Lines 71 to 121 perform the same tasks as lines 68 to 111 of Code 6.1 with the major difference found in line 94 where the computations for $\mathbf{C}(i, l\_col)$ happen if and only if `l_col` is encountered in `stack` of the CRS of $\underline{\mathbf{C}}$. We use binary search to find `l_col` in `stack`; this is the reason for which `stack` needs to be sorted before.

To extend the implementation of $\underline{\mathbf{C}}$ to the nonblocking backend, we follow the same ideas detailed in section 5.3.

Chapter 6

Numerical experiments

To test the performance of the implementation of `grb::mxm_masked` in the nonblocking backend, we compute $\mathrm{tr}\left(\mathbf{A}^3\right)/6$ for the adjacency matrices shown in Table 6.1, and compare to the performance of the reference backend for the same operation. Code 6.1 shows what primitives are requiered to perform that operation.

```
int main( int argc , char ** argv )
{
    // initialization A, C, D, D_mask
    // ...

    // fill matrices A and D_mask with data
    // ...

    // initizaliton of ring and monoid
    const Semiring< grb::operators::add< double >, grb::operators::mul< double >,
        grb::identities::zero , grb::identities::one > ring;

    const Monoid< grb::operators::add< double >, grb::identities::zero > monoid;

    // T = A*A. This primitives do not lead to pipeline execution
    grb::mxm(C, A, A,ring, RESIZE);
    grb::mxm(C, A, A,ring);

    // computation of masked D = T *C.
    // D_mask is a diagonal matrix. These primitives do not lead to pipeline
        execution
    grb::mxm_masked( D, D_mask , C, A, ring , RESIZE );
    grb::mxm_masked( D, D_mask , C, A, ring );

    // Computation of tr(A^3). This primitive leads to the pipeline execution
    s = 0;
    grb::foldl( s, D, monoid );

    // computation of tr(A^3)/6
    s = s/6;

    // ...
}
```

Code 6.1: Computation of $\mathrm{tr}\left(\mathbf{A}^3\right)/6$ in ALP/GraphBLAS.

From the data dependence analysis discussed in section 4.3.3, for Code 6.1, a single pipeline is constructed since all the primitives share data, and the pipeline is executed by the primitive in line 25 since it returns a scalar. The same Code 4.3.3 is used for the reference and nonblocking backends. For example, to compile that code using the reference backend, one types on the terminal `make trace_A_cube_reference`, which instructs the compiler to use the primitives of implemented in the reference backend.

The tests were run on one node of the cluster with the following characteristics: a dual-socket system with two ARM920 processors (architecture aarch64) for a total of 96 cores at maximum frequency of 2.6GHz per core. The main capacity of the main memory is 503 GB, L1d cache memory is 6 MiB, L1i cache memory is 6 MiB, L2 cache memory is 48 MiB, and L2 cache memory is 96 MiB. For multi-threaded execution, all experiments are run on the same socket. Operating system is Ubuntu 20.04.5 LTS, kernel 5.4.0-148-generic. The code was compiled using g++ 9.4.0.

All matrices in Table 6.1 can be found on `http://sparse.tamu.edu/`; these are square matrices that represent the adjacency matrices of undirected graphs and density of nonzeros represents the percentage $\frac{\#\text{ nnz}}{\#\text{rows}^2} * 100\%$.

matrix ID	Matrix name	# rows	nnz	Density of nonzeros (%)	# triangles
1	coPapersDBLP	540,486	30,491,458	1.04E-02	444095058
2	europe_osm	50,912,018	108,109,320	4.17E-06	61710

Table 6.1: Matrices for tests.

The number of triangles shown in the last column of Table 6.1 are reported in TABLE II in [1] and these coincide with the number of nonzeros we obtain during all experiments for each matrix.

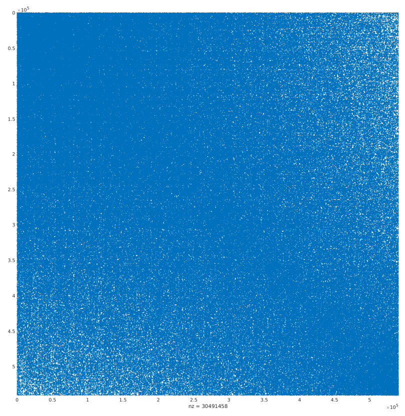


Figure 6.1: Sparsity patter of matrix 1 in Table 6.1.

The experiments for matrix coPapersDBLP are as follows. Firstly, we utilize 48 threads for the nonblocking backend, and use a tile size of 128. Then we run the Code shown in 6.1 20 times. The value reported in Figure 6.2 (first bar char to the left) is the average value of those 20 runs. We measure time after all matrices have been initialized and filled with data. We repeat the same procedure for tile size equals to 256 and 512. Subsequently, we repeat the same process for number of threads equal to 24 and 12. In addition, the same figure displays the computing time obtained when using the reference backend. For this backend, we run 20 runs of Code 6.1 and report the average value in Figure 6.2. In Figure 6.2, the computing time corresponds to

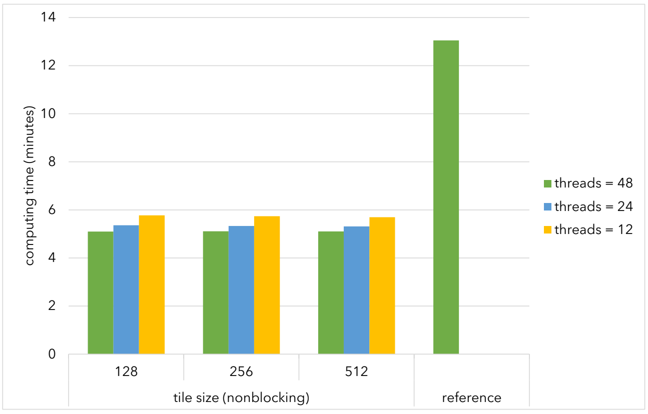


Figure 6.2: Computing times for matrix 1 in Table 6.1.

the time it takes to execute the primitives shown in Code 6.1; this time does not consider the time it takes to load data into memory.

We observe in Figure that 6.2 that the fastest computations are reached when using 48 threads. Importantly, in the results of the nonblocking backend, we observe that there is no a significant differences between the performance when varying the number of threads and the tile sizes. Figure 6.1 help us understand the reason behind this situation. Indeed, let us say that we select a tile size of 128; then if matrix coPapersDBLP is divided into that number of tiles, we will observe that each tile works or receives a similar quantity of nonzeros. That is, the data balance of matrix coPapersDBLP is homogeneous. For this matrix, the execution of Code 6.1 in the nonblocking backend for 48 threads is 2.5% faster than the execution of the same code in the reference backend. In this case, we may argue that the data reusage on cache memory and parallel execution, both core ideas of nonblocking execution, have led to performance improvements.

From Table 6.1, we observer that even if the matrix coPapersDBLP contains a significant number of nonzeros - more than 99% of its elements are nonzero, the nonblocking backend leads to having performance improvements over the reference backend.

On the other hand, results for matrix europe_osm in Table 6.1 when running Code 6.1 in the nonblocking backend show a clear performance degradation with respect to the reference backend's. We utilize 96 threads only for these experiments and two tile sizes 4098 and 2048; we run 10 runs for each tile size and report the average computing time for each tile size in Figure 6.3. For the reference backend, we run 30 experiments; we report the average computing time. On average, each run in the reference backend take one minute.

In Figure 6.3, the performance degradation of Code 6.1 when executed in the nonblocking backend is explained mainly by the fact that matrix europe_osm has too many nonzeros for a matrix of its size; that is, for one nonzero, it has a million nonzeros.

For example, let us take a tile size of 4098, then when tiling, matrix europe_osm will be divided into 12429 tiles, which corresponds to each thread computing 130 tiles. In this case, assuming the nonzeros are distributed homogeneously in that matrix, each tile will have one nonzero for a million nonzeros. Moreover, for each tile a set of lambda functions is executed. Each of the primitives in Code 6.1 corresponds to three lambda functions in the computational phase (`grb::foldl` also consists of three lambda functions even if it does not have any symbolic phase). This means that during the execution of the pipeline, around 111,861 lambda functions are called. Consequently, a lambda function is called for every 1000 nonzeros.

Then, such performance degradation is expected. Firstly, the overhead of calling the lambda functions shadows all potential performance improvements brought by the nonblocking backend. We can think of this as the fact that data re-usage in this case does not lead to any performance improvement with respect to the reference backend because there is hardly data to reuse on cache memory in comparison to the cost of calling such a large number of lambda functions. For matrix europe_osm, nonblocking execution significantly worsens the computing time of Code 6.1 compared to the reference backend.

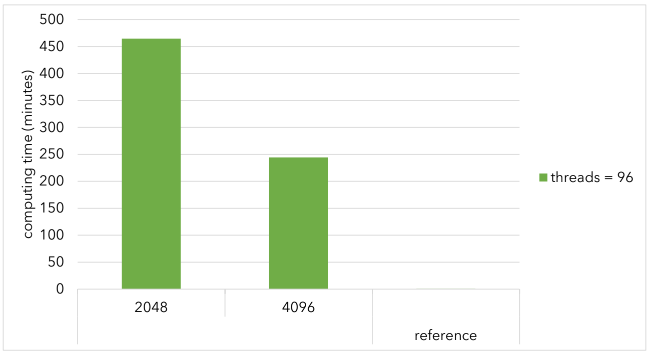


Figure 6.3: Computing times for matrix 2 in Table 6.1.

In addition, memory consumption in the nonblocking backend increases significantly for matrix europe_osm. Even if for each thread available on the system, a SPA is created and stored before the execution of the pipeline for the output matrix of `grb::mxm`, such memory is allocated on the heap by using `STL` vectors and then this may fragmented due to the size of the matrix. In the current implementation of the nonblocking execution of level-3 primitives, we do not perform any memory fragmentation check. Since each tile uses its own SPA from the ones available, one consequence of this potential segmentation is that tiles can require a significant amount of time to read from and work on their SPA. For instance, since europe_osm is the adjacency matrix of an undirected graph, each of its occupied positions are 1's; for each 1, 8 bytes of main memory are used; then, a SPA created per thread requires $8 \times$ (#rows in europe_osm) $\times\ 3$ bytes of memory, which is around 1.2 GB. Assuming that 96 threads are utilized, approximately 120 GB of extra memory are required for executing Code 6.1 in the nonblocking backend. Conversely, there is a single SPA for the output matrix of `grb::mxm` in the reference backend.

Finally, the current implementation of the `grb::mxm` repeats twice a loop over the nonzero elements of the input matrices during the computational phase, which can also explain the important performance degradation for matrix europe_osm.

Chapter 7

Conclusions and future work

Currently, more tests are being performed to tests the performance of `grb::mxm` in the nonblocking backend for other algorithms apart from the one presented in Code 6.1. From the presented experiments with two matrices, we may state the the nonblocking execution of `grb::mxm` leads to performance improvement but the density of nonzeros in the input matrices plays an important role in this improvement.

The core idea of nonblocking is to reuse data on cache memory. However, the nonblocking execution of `grb::mxm` for large matrices with too many nonzeros resulted in substantial performance degradation since data reusage and parallel execution do not compensate the overhead of calling lambda functions for each tile of the output matrix of `grb::mxm`. The overhead of utilizing lambda functions for a tile that has few nonzeros is more significant that the benefits of reusing data on cache.

Concerning future work, firstly, we could think of implementing level-3 primitives in the reference and nonblocking backends as in-place operations as indicated in the GraphBLAS API C.

Concerning the `grb::mxm` primitive, it is worth mentioning that utilizing the same algorithm to perform that operation in the reference and nonblocking backends is a good idea from a software development perspective since having a unique algorithm allows for a clear and fairer comparison between the performance in both backends. Additionally, a strong reason in favor of utilizing the same algorithm for `grb::mxm` in both backends is that it meshes well with the sparse vector-sparse matrix product algorithm in ALP/GraphBLAS. However, at the same time, this prevents testing or trying different algorithms that may be better for a specific backend. Then, an important direction of research is to investigate new algorithms to perform the spM-spM product in the nonblocking backend.

In terms of the nonblocking implementation of `grb::mxm`, masked and unmasked, and `grb::eWiseApply` presented in this work, the SPA that each tile uses during the computational phases is implemented using `STL` vectors. Nevertheless, in ALP/GraphBLAS, the CRS and CCS formats, and the SPA of matrices are stored using either the `numa_alloc_interleaved` or `posix_memalign` functions from the `STL` library, which allows better memory usage due to the fact that a memory alignment requirement can be ensured using the latter. Then, further work is to be done in the nonblocking backend in terms of memory management for the SPAs to which tiles access during the computational phase.

Moreover, the implementation in the nonblocking backend of `grb::mxm` is to be tested within a program where level-2 primitives are used. This may highlight how a complete nonblocking implementation performs. Related to this point, testing the nonblocking implementation of the level-3 primitives in a program such that the pipelines contain more than three stages is to be performed. For example, for the tests presented in this work, the pipeline constructed when running Code 6.1 contains three stages only.

In addition, there is need for developing an analytic model to select the tile sizes in level-3 primitives automatically for each pipeline at run-time in the nonblocking backend, as presented in [8] for level-2 primitives.

Bibliography

[1] Ariful Azad, Aydin Buluç, and John Gilbert. “Parallel triangle counting and enumeration using matrix algebra”. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE. 2015, pp. 804–811.

[2] Aydin Buluç et al. “Design of the GraphBLAS API for C”. In: *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2017, pp. 643–652.

[3] Andrew Duncan. “Powers of the adjacency matrix and the walk matrix”. In: (2004).

[4] Walter Gander, Martin J Gander, and Felix Kwok. *Scientific computing-An introduction using Maple and MATLAB*. Vol. 11. Springer Science & Business, 2014.

[5] John R Gilbert, Cleve Moler, and Robert Schreiber. “Sparse matrices in MATLAB: Design and implementation”. In: *SIAM journal on matrix analysis and applications* 13.1 (1992), pp. 333–356.

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

[7] Bernhard H Korte et al. *Combinatorial optimization*. Vol. 1. Springer, 2011.

[8] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan N Yzelman. “Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance”. In: *ACM Transactions on Architecture and Code Optimization* 20.1 (2022), pp. 1–23.

[9] Nazanin Movarraei and MM Shikare. “On the number of paths of lengths 3 and 4 in a graph”. In: *International Journal of Applied Mathematics Research* 3.2 (2014), p. 178.

[10] Nitish Srivastava et al. “Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 766–780.

[11] AN Yzelman et al. “A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation”. In: *Preprint* (2020).