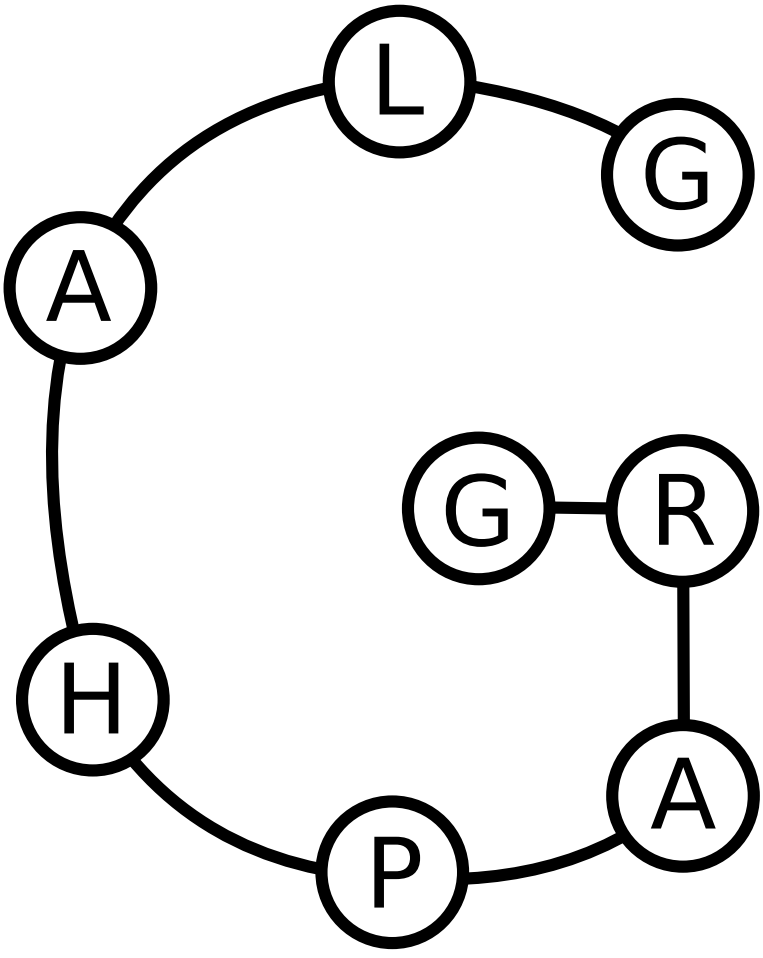
Efficient Execution of User-Provided Graph Algorithms in a Graph Database

Version of August 28, 2023



Daan de Graaf

Efficient Execution of User-Provided Graph Algorithms in a Graph Database

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Daan de Graaf
born in Utrecht, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: The Graphalg logo.

Efficient Execution of User-Provided Graph Algorithms in a Graph Database

Author: Daan de Graaf
Student id: 5638976

Abstract

Graph databases are systems to efficiently store and query large graphs. As graph databases grow in popularity, they are used to answer increasingly diverse and complex queries. However, graph databases typically have a very limited query language that cannot express arbitrary algorithms. As a result, many users treat the database as a storage layer to export data from and develop algorithms in external tools, wasting computation power and storage space.

We present graphalg, a high-level, domain-specific language for writing graph algorithms embedded into traditional graph queries. Our language is based on linear algebra, with a syntax resembling GraphBLAS, and implemented in the AvantGraph database.

We implement a compiler for graphalg that can target an interpreter built on top of a GraphBLAS implementation. Alternatively, our compiler can transform graphalg programs into a relational algebra with loops, unifying the representation of query and algorithm. We evaluate the programmability and performance of our system on the GAP Benchmark Suite for graph algorithms. Our language is expressive enough to concisely represent all GAP benchmark programs, with the majority of programs achieving performance comparable to an optimized C implementation.

We conclude that graph algorithm support can be integrated into graph databases to increase their programmability. Running graph algorithms inside of the database increases performance and reduces memory consumption compared to using external tools for the analysis. Rather than thinking of graph databases as limited tools for answering simple queries, we demonstrate that they can instead be a programmable framework for efficient large-scale data analysis.

Thesis Committee:

Chair:	Prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
Daily supervisor:	Dr. S. Chakraborty, Faculty EEMCS, TU Delft
Computer Engineering member:	Dr. Z. Al-Ars, Faculty EEMCS, TU Delft
External supervisor:	Dr. N. Yakovets, Faculty WSK&I, TU Eindhoven

Preface

I would like to start this thesis by thanking my friends, family and colleagues for their help and support during this project. This thesis may bear my name, but it would not have been possible without you. My heartfelt gratitude goes out to these people in particular:

To **Soham**, for agreeing to supervise this project. Your expertise in compilers has been invaluable, as was your general advice on doing research and writing it down. It was a pleasure to work with you, and I hope we can collaborate in future research.

To **Nick**, for pitching the idea that sparked this thesis. You taught me how to build a database, and how to sell it. I look forward to starting my Ph.D. with you as my co-promotor.

To **Bram**, for your help debugging my horrible code. I promise that one day, I will write those unit tests.

To my dear girlfriend **Margot**, who always found new ways to distract me from work. Thank you for your unwavering love and support during these months.

To my loving parents **Wouter** and **Kita**, who taught me to work hard for the things you care about, and to do it your way.

To **Laura** and **Jasper**, for the coffees we shared discussing our theses. I have no doubt you will deliver excellent theses.

To my new **colleagues at the TU/e Database Group**, for welcoming me into their team. Though it has hardly been two months, I already feel right at home.

To the **thesis committee**, for taking the time out of your busy schedules to review my work and offer your feedback. I strive to make it worth your while.

Daan de Graaf
Eindhoven, the Netherlands
August 28, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are used to model a wide variety of systems across disciplines. They are the obvious way to represent a network of connected devices, rail infrastructure or an electrical grid. Algorithms over graphs can be used to perform such diverse tasks as finding the most influential people in a group, uncovering fraudulent wire transfers, or finding the shortest path between two points.

Systems specifically designed to store and analyze large graphs, *graph databases*, have been developed to perform all these tasks. Graph databases typically offer a declarative interface for the user to specify search queries, which the database efficiently executes on the large graph it stores.

Support for graph queries in databases is becoming mainstream. Database vendors such as Oracle [43] and Microsoft [32] provide graph query support in their offerings. Furthermore, the 2023 SQL standard will include a graph query syntax extension [18], allowing vendor-neutral graph queries embedded in SQL.

The query languages offered by graph databases are usually declarative and focused on pattern matching. Database vendors recognize a user demand to perform complex analyses that cannot be expressed as simple pattern matching [39, 7]. Their solution to this problem is to provide users with a library of algorithms for common use cases such as centrality metrics, community detection and pathfinding. If however, a user needs an algorithm that the vendor does not provide an implementation for, they cannot provide a custom algorithm implementation and run it inside the graph database. Their only option is to export the graph and run the analysis in an external tool. Exporting large volumes of data is expensive both in computing and storage costs. Furthermore, external tools cannot access the statistics the database maintains about the graph, leading to a suboptimal processing pipeline. Due to its restricted functionality, the graph database is demoted to a mere storage layer in the analytics pipeline.

1.1 Contributions

We posit that this is not a fundamental limitation of graph databases, but rather a by-product of the limited programmability of their interface.

Our contributions are the following:

- We propose *graphalg*, a domain-specific language (DSL) for writing algorithms that can be executed directly by a graph database (Chapter 3). Like most database query languages, graphalg consists of high-level operations with semantics that are easy to reason about and can be optimized well by a query planner. Graphalg natively supports iteration and conditional branching, and can concisely express a wide variety of algorithms. Graphs are represented as matrices, with operations on them expressed in linear algebra.

Graphalg is inspired by the GraphBLAS library [12], an open API of standard components for building algorithms in the language of linear algebra. While the use of linear algebra to implement graph algorithms is a proven approach, we are the first to present a language specifically designed for writing algorithms in this style, and the first to integrate it into a graph database.

- We develop a compiler with two backends based on different execution models, with shared initial phases for parsing and normalization (Chapter 4). We provide a detailed description of the intermediate representations and transformations that make up the system and highlight the differences between the two backends. We also cover the integration of the compiler into the target database, AvantGraph [30].
- Our GraphBLAS backend (Chapter 5) compiles graph algorithms into GraphBLAS library calls and executes them in an interpreter. We show how programs written in our high-level DSL can be optimized to run as fast as a C implementation by fusing operations and automatically allocating buffers for intermediate results.
- Additionally, we present the 'operators' backend, which transforms graphalg programs into extended relational algebra (Chapter 6). We show how programs written in our DSL can be transformed into the internal IR of AvantGraph, fully integrated into queries, and executed by the same runtime. We discuss what additional operators are needed to support iterative algorithms with linear algebra operations, most notably loops and semirings, and how to implement them.

The remainder of the thesis is organized as follows:

- In Chapter 7, we evaluate our system based on the design goals stated in Section 3.1, with a particular focus on efficiency and expressivity. Execution time and memory usage are the key metrics to determine the success of our approach. We use the input graphs and kernels specified in the GAP Benchmark Suite [3].
- Our work overlaps with a considerable volume of existing research into various topics such as algorithm support in databases, languages for writing graph algorithms and linear algebra primitives in databases. A review of related work and how it compares to our approach is in chapter Chapter 8.
- We discuss the limitations and future work in Chapter 9, and finally conclude with Chapter 10.

For readers unfamiliar with (graph) databases or compilers, we provide background information on those topics in Chapter 2.

We expect that graphalg will be released under an open source license as a part of AvantGraph in late 2023. Until then, we provide a binary distribution of AvantGraph with graphalg support for experimentation and reproducibility of our results. The artifact is available at `https://doi.org/10.5281/zenodo.8286432`.

Chapter 2

Background

For this project, we develop a compiler and integrate it into a graph database. In this chapter, we cover the relevant topics from the compiler and database literature, notably:

- Graph databases as seen from the user perspective (Section 2.1).
- Database internals, in particular query planning (Section 2.2) and query execution (Section 2.3)
- Program representations in a compiler (Section 2.4)
- Program transformation using rewrite rules (Section 2.5)

See Figure 2.1 for a high-level view of the query lifecycle in a graph database. The query planner, execution plan generator and execution engine are particularly important to our work. They are covered in detail in Section 2.2 and Section 2.3.

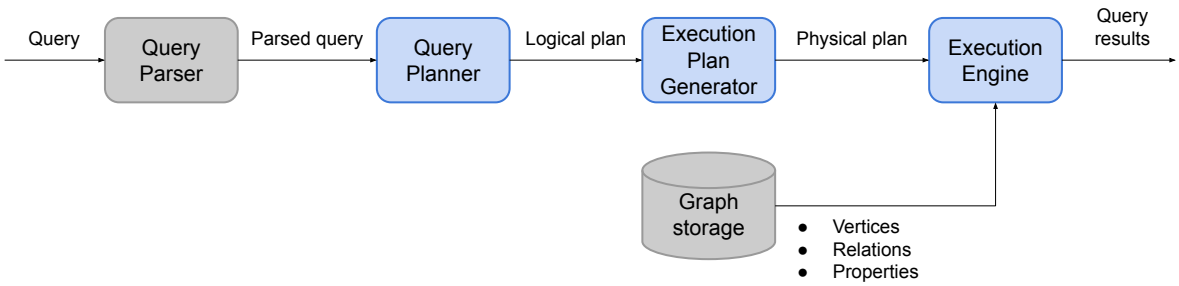


Figure 2.1: Typical architecture of a graph database.

In Figure 2.2 we show a similar overview, this time of a compilation pipeline. The parser is the only part of the pipeline that we do not cover in this chapter since it is autogenerated using ANTLR[41]. Representations used between the different stages are presented in Section 2.4, while the stages themselves are discussed in Section 2.5.

Figure 2.2: Typical architecture of a compiler.

2.1 Graph Databases

2.1.1 Data Model

Graph databases store information as a network of vertices connected via edges. For certain problem domains, this is a more natural model than the traditional relational database model of tables and columns. Consider for example a database that tracks the friendships that exist between individuals in a community. In a relational database, we can model this using a `Person` and a `Friendship` table, as shown in Table 2.3.

(a) Person table

id	name
1	Alice
2	Bob
3	Charlie

(b) Friendship table

person_id	friend_id
1	2
1	3
2	1
2	3
3	1
3	2

Table 2.3: Friendship database in tabular form.

The properties of every person in the database, only their name in this example, are stored in Table 2.3a. Friendships are tracked in Table 2.3b by linking two person ids, e.g. the first row states that person 1 (Alice), is friends with 2 (Bob).

While representing this community in tabular form is possible, a graph database provides a more intuitive model. Every person in the community can be presented with a unique vertex, and we can add an edge between two vertices if they are friends. Plain vertices and edges are already sufficient to model the Friendship table and the id column of the Person table, but the database must also store the name of each person (or vertex). Many graph databases, including AvantGraph, allow associating key-value pairs with vertices and edges. This is the *property graph model*, and it lets us attach properties such as `name:'Alice'` to vertices. Figure Figure 2.4 shows the same friendship database represented as a property graph.

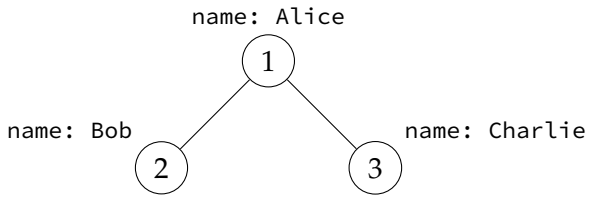


Figure 2.4: Friendship database in property graph form.

While it may not be apparent from this simple example, the property graph model is highly flexible:

- Vertices may have many different properties, each with a different key. Properties can be added to a subset of vertices, unlike the relational model where adding a column applies to every row in the table.
- Edges can have properties too, creating a weighted graph. In a road network, for example, this can be used to store the distance between locations or a speed limit.
- Edges can be labeled to store multiple relations in a single graph. Next to the edges in Figure 2.4 that track friendships, we could for example introduce a 'family' relationship. Edge labels are conventionally written in uppercase, so an appropriate name for the

relation in Figure 2.4 would be `FRIEND`. Vertices are not exclusive to a relationship, e.g. two vertices connected via a `FRIEND` edge may also have a `FAMILY` edge.

Having covered the property graph data model used in graph databases like AvantGraph, let us consider how to query such graphs.

2.1.2 Querying graphs

AvantGraph (and many other graph databases) support the OpenCypher query language [40], a declarative language inspired by SQL. OpenCypher focuses on *subgraph matching*: the user specifies a target pattern, and the query engine searches the graph to find subgraphs that match the given pattern.

To use the graph from Figure 2.4 as an example, consider that a user may wish to retrieve the names of all individuals that are friends with Alice. This query is visualized by Figure 2.5.

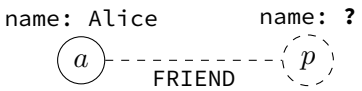


Figure 2.5: Query pattern to find names of Alice's friends.

Vertex a matches any vertex with the value 'Alice' for the `name` property, and p matches any vertex with a `FRIEND` edge to a valid a. This pattern has a direct encoding in OpenCypher, shown in Listing 2.1.

Listing 2.1: OpenCypher query to find names of Alice's friends

```
MATCH (:Person {name: 'Alice'}) - [:FRIEND] -> (p:Person)
RETURN p.name
```

Not all queries can be expressed as subgraph matching. For example, a user may wish to find the shortest path between Alice and Bob. In the graph of Figure 2.4 this is trivial, but in a large graph there may be many paths between the two vertices, and the database must return the shortest of all paths. OpenCypher supports such queries by providing built-in functions for common algorithms. For this particular query, the `shortestPath` function can be used, as shown in Listing 2.2.

Listing 2.2: Query to find the shortest path between Alice and Bob

```
MATCH
 (Alice:Person {name: 'Alice'}),
 (Bob:Person {name: 'Bob'}),
 p = shortestPath( (Alice)-[:FRIEND*]-(Bob) )
RETURN p;
```

The function `shortestPath` evaluates all paths that satisfy the pattern `(Alice)-[:FRIEND*]-(Bob)`, and returns the shortest one. Invoking `shortestPath` inside of **`MATCH`** allows binding it to a variable, which is appropriate for a function that returns a single result. If however the user requires an algorithm that returns multiple results, it must be invoked through OpenCypher's procedure syntax instead. We show an example of a procedure call in Listing 2.3.

Listing 2.3: Calling a procedure to find shortest paths to all vertices from a single source

```
MATCH (source:Person {name: 'Alice'})
CALL allShortestPaths(source)
YIELD nodeId, length
RETURN nodeId, length
```

The above query returns the length of the shortest path to every reachable vertex in the graph, starting from the given source. The **CALL** clause defines the algorithm to be executed and the parameters to pass. Procedures return a stream of tuples. To control which columns of the stream are passed on to later clauses, **YIELD** is used to select the desired columns.

Most of the programs that we use to evaluate the system return multiple results, so we focus on supporting procedures in this work.

2.2 Query Planning

After the database has parsed and validated a query, it must decide how to compute the desired result. It is the job of the *query planner*, sometimes also referred to as *query optimizer*, to find the most efficient execution plan that answers the query.

2.2.1 Relational Algebra

The first step of query planning is to convert the parsed OpenCypher into a more regular structure that is easy to manipulate and reason about. Like most query planners, AvantGraph uses *relational algebra* to represent queries. In relational algebra, queries are represented as a tree of operators. An operator receives tuples from its child operators, manipulates them in some way, and passes on the result to its parent operator. The tree of operators thus defines a data processing pipeline, which can perform a complex computation by composing many simple operators.

To give an example of a query converted to relational algebra, let us revisit the query in Listing 2.1. Because relational algebra has a more straightforward mapping to SQL, we also give the equivalent SQL query (based on the schema in Table 2.3), shown in Listing 2.4

Listing 2.4: SQL query to find names of Alice's friends.

```
SELECT p.name
FROM Person AS a
JOIN Friendship AS f ON (a.id = f.person_id)
JOIN Person AS p ON (p.id = f.friend_id)
WHERE a.name = 'Alice'
```

In Figure 2.6 one possible mapping to relation algebra is shown.

The leaves of the tree are references to the tables of Table 2.3, while the internal nodes are operators. To identify particular elements of tuples we assign a single-letter label to each table reference (`a`, `f` and `p`), and use the convention *table.column* to refer to a column. The semantics of the operators are as follows:

- Π is the projection operator. It forwards the given subset of the input columns.
- σ is the selection operator. It forwards only the tuples for which the predicate holds.
- $\bowtie$ is the join operator. It computes the cartesian product between all tuples in the two input relations and outputs all combinations for which the given predicate matches. Columns of both inputs are available in the output.

With the semantics of each of its operators defined, the query as expressed in relational algebra becomes a plan for computing the answer to the query. We refer to it as a *query plan*.

2.2.2 Finding a better plan

The query plan of Figure 2.6 is but one possible plan to answer this query. This is closely related to the declarative design of the query language: the user specifies only what is to be

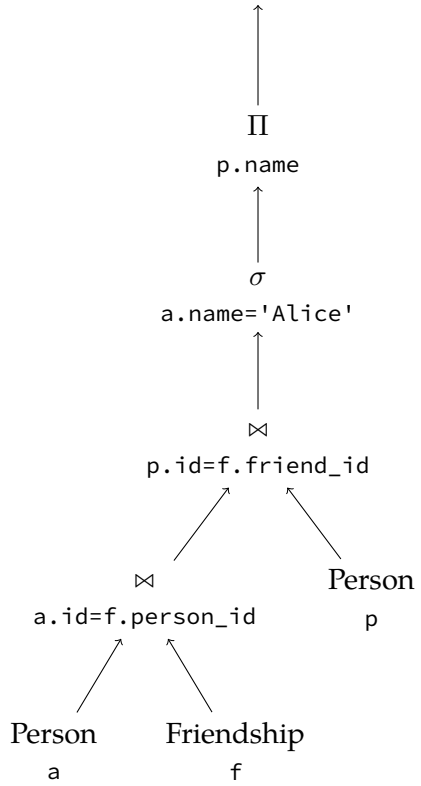


Figure 2.6: Query plan to find names of Alice's friends, in relational algebra.

computed, instead of how it should be done. For most queries, there will be many different query plans that produce identical results, but the way they compute those results differs.

The database system is free to choose any plan for a particular query, as long as it is correct, so it can evaluate many plans and pick the best one according to some cost function. Users typically prefer their queries to be answered quickly, so most databases optimize for a low execution time and low memory usage to allow running many queries concurrently. For a discussion on how to maintain the correctness of the query plan during optimization, see Subsection 2.5.1.

To keep the following example simple, we optimize for the lowest number of tuples passed between operators. According to this cost metric, the plan in Figure 2.6 is far from optimal. The σ operator is only applied after all joins have been performed. Applying this selection closer to the leaves does not change the result, and it reduces the number of tuples passed upstream. This particular optimization is common in databases and is referred to as *predicate pushdown*. In Figure 2.7 we show an alternative query plan that is more efficient according to our defined cost function.

AvantGraph's query planner includes predicate pushdown and other simple optimization rules. Additionally, it has a plan enumerator that quickly constructs thousands of equivalent query plans, and picks the best one according to the estimated cost.

2.2.3 Logical to Physical plan

Initially, the planner considers pure relational algebra operators, without a particular implementation. The plans it operates on at this stage are called *logical query plans*. Once logical planning is complete, the database decides how to access the input tables (i.e. their physical representation on disk) and what implementations should be used for particular operators. This creates a *physical* plan determining exactly how the query will be executed.

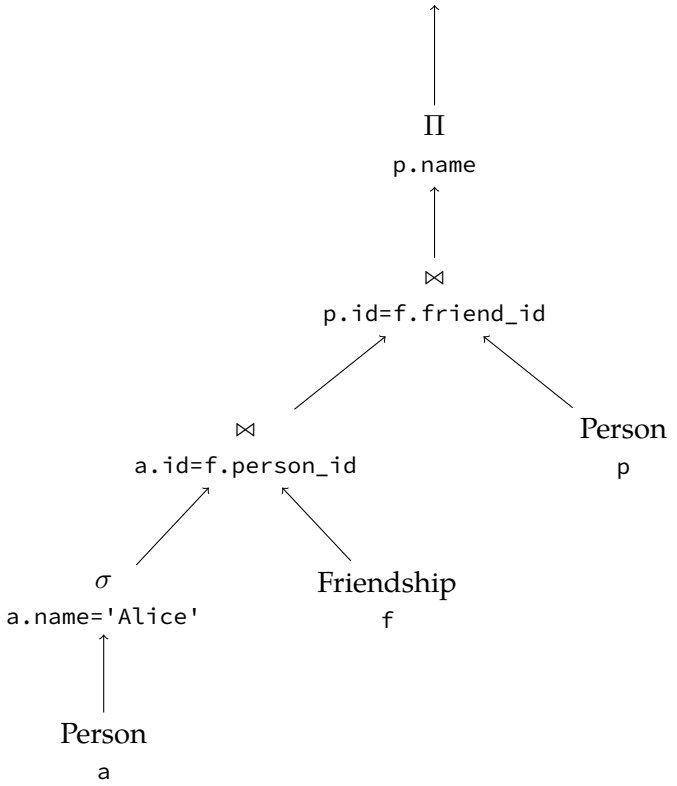


Figure 2.7: Query plan to find names of Alice's friends, with σ pushed down.

A join operator, for example, could be implemented in many different ways:

- Nested loop join: for each tuple in the first input, loop over all tuples in the second input.
- Hash join: build a hash table of one of the inputs (with a key based on the columns in the predicate) and probe it for every tuple in the other input.
- Sort-merge join: sort both inputs on the columns in the predicate, then do one simultaneous pass over both inputs.

None of these joins is strictly better than the others, but a query planner can estimate which join implementation will be most efficient in a particular part of the query, and select it for the physical plan.

In AvantGraph, the logical plan is stored in the *Intermediate Plan Representation* (IPR). IPR supports annotations to indicate a particular join implementation or access method, blurring the lines between logical and physical plan, but it is otherwise a textbook logical plan representation. Physical plans are referred to as *Execution plans* in AvantGraph internally. Execution plans are generated as the final step in query planning and are directly executed by the runtime, without further modification.

2.3 Query Execution

After the query planner has established a physical plan, the *query execution engine* executes it to produce the query results. In the literature, this phase is also referred to as *query processing*.

2.3.1 Volcano Model

The classical execution model is the Volcano model [20]. Here, operators are represented as iterators, producing one tuple at a time. The execution engine repeatedly polls the root node in the plan for the next tuple, and operators may poll their child operators for input tuples if necessary. By representing operators as iterators, we avoid having to buffer large intermediate outputs between operators. If all the operators in a subtree can apply their transformations locally per tuple, i.e. they do not need to combine multiple tuples, that subtree will be *pipelined*: The operators are applied successively without any buffering in between. The implementation may even be able to keep the tuple in machine registers between operators, allowing for very fast execution with minimal memory accesses.

As an example, consider a query plan in Figure 2.8. If the join of L and R has many results, but there are only a few tuples for which $L.a + R.b > 0$ holds, an approach without pipelining would have to first materialize this large intermediate result, then filter out almost all tuples. With pipelining however, the selection predicate can be directly applied to tuples as they are produced by the join, so the pipelined execution engine will produce the first tuples sooner and use significantly less memory to process the query.

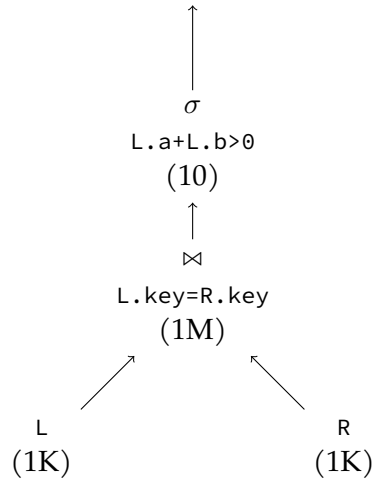


Figure 2.8: A query plan with an expensive intermediate. Each operator is annotated with the number of tuples it produces. All tuples in L and R have the same value for column `key`, so the join is a cartesian product.

AvantGraph makes three key changes to the volcano model:

- Vectorized execution: operators pass blocks of tuples in a columnar format.
- Push-based, multi-threaded evaluation: operators run in parallel, pushing data to their parents.
- Relations between operators: handles data fan-in and fan-out between operators.

2.3.2 Vectorized Execution

In AvantGraph, the unit of data is a block of tuples rather than a single tuple. Blocks are laid out in columnar format, with separate data arrays per output column. One reason for this layout is that it allows the operator implementation to load columns into wide SIMD registers and process the data with vector instructions available on modern CPUs.

Even if vector instructions cannot be used, a columnar representation can still speed up query execution by reducing the amount of data to read. Blocks may contain many columns,

but most operators only access a few of them. If the tuples were laid out sequentially in memory, the entire block would have to be loaded from the main memory into the cache, including the columns that are not needed. In a columnar representation, however, we can selectively load only those columns that are needed by the operators. An example of a block of 4 tuples with 4 columns is given in Figure 2.9.

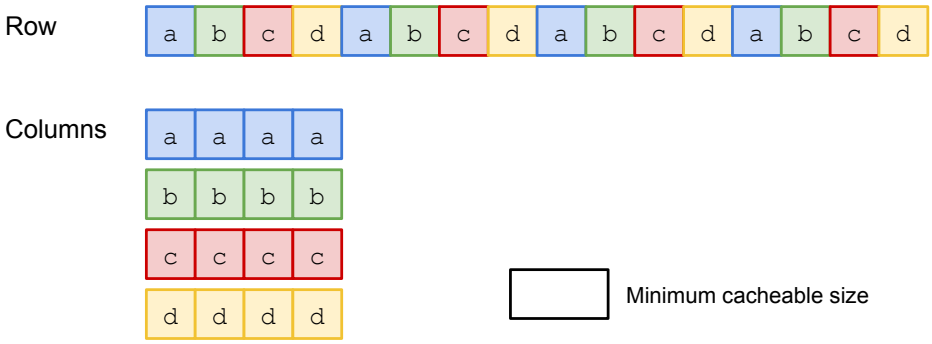


Figure 2.9: Row vs. column-based storage. The minimum cacheable size is two fields.

Consider a selection operator with predicate $a = c$. If the tuples are stored sequentially in memory, reading any a will also bring a b into the cache, and similarly reading c brings in d. The full block will be loaded into the cache, using 8 cache lines. If, however, we assume columnar storage, the b or d values can be skipped entirely. Only the a and c columns are read into the cache using 4 cache lines.

2.3.3 Multi-threaded execution

While vectorized execution helps process more data per CPU core, a fast execution engine is not constrained to a single core. On modern CPUs with many cores, multi-threaded execution can speed up queries dramatically. The AvantGraph runtime maintains a thread pool with one thread per core and has a custom scheduler that distributes tasks to available worker threads. Operators do not directly process data blocks but rather submit work units to the scheduler which will run at the scheduler's instruction.

If there are available threads with no work units to run, the scheduler polls all operators in the query plan for new work units. There is no requirement that a parent operator has requested tuples for an operator to start performing work, as in the volcano model. Instead, all operators can produce work units at any time, allowing for maximum parallelism. When an operator produces a block, it is pushed to its output relation (relations are discussed in the next section) and the block is buffered there until its parent is ready to process it. AvantGraph's execution strategy is therefore defined as *push-based* [47], rather than the *pull-based* strategy seen in the Volcano model.

The leaf operators in the query, which read the graph, will always be the first operators to schedule work units since they do not have input dependencies. Once they start to produce inputs for their parents, those operators can also begin processing blocks and trigger their parents to start scheduling work units. Evaluation of the query thus starts at the bottom of the plan and bubbles up to the top, so it is sometimes also referred to as *bottom-up* evaluation.

As an example, consider the query plan shown in Figure 2.10.

For this query, the 'Person' table refers to the table given in Table 2.3a. In Figure 2.11 we see how this query executes in a pull model. When the same query is executed in a push-based engine, the call direction is inverted, as shown in Figure 2.12. To simplify the figures, the operators are connected directly to each other. AvantGraph instead puts *relation* objects between operators, which we discuss in the next section.

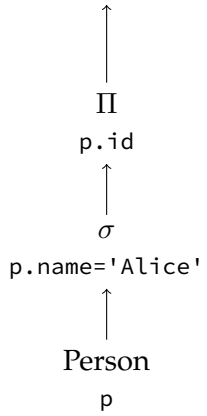


Figure 2.10: Query plan to find the ID for Alice.

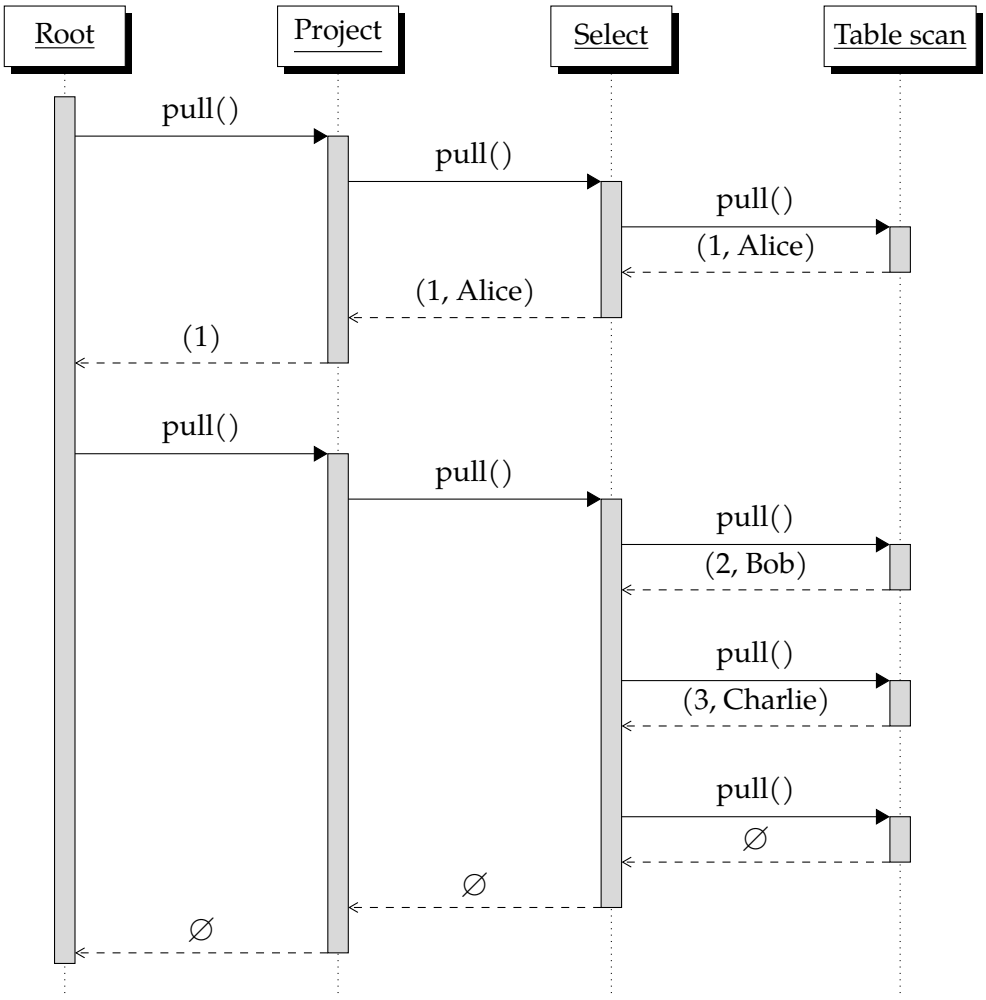


Figure 2.11: Execution of a simple select and project query using the pull model.

2.3.4 Relations

In AvantGraph execution plans, operators are not connected directly to each other. Instead, a child operator sends its output to a *relation*, which in turn is the input to the parent operator. Relations act as a buffer between operators, storing received blocks until an input operator is ready to process them. Relations can combine the outputs of multiple operators into a single stream of blocks (fan-in), or send the same blocks to multiple operators (fan-out). An

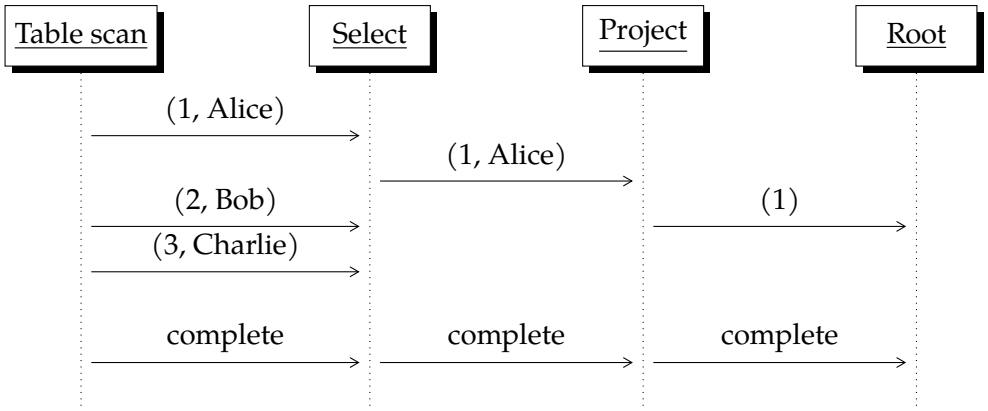


Figure 2.12: Execution of a simple select and project query using the push model.

example relation combining fan-in and fan-out is shown in Figure 2.13.

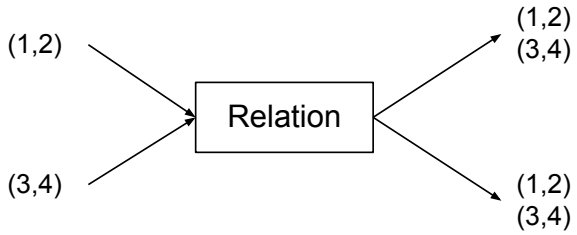


Figure 2.13: A relation combining multiple inputs (fan-in) and distributing them to multiple outputs (fan-out).

2.4 Program Representation

While our compiler reads programs in textual format, internally it uses a different representation that is easier to manipulate and analyze, much like a query planner converts queries into relational algebra. Because this representation is different from the original input form, and not preserved in the final output, it is an *intermediate representation* (IR). Our IR combines the following concepts and tools:

- Abstract Syntax Trees [1]: the program text is converted into a tree structure.
- Static single assignment [11]: all variables in the abstract representation are assigned exactly once.
- We use the MLIR framework [29] to define our IR.

2.4.1 Abstract Syntax Trees

An *abstract syntax tree* (AST) is the classical representation for programs in a compiler. When converted to a structured tree, programs are easier to analyze and manipulate compared to their textual form. Take for example the expression $1 + a \times 2$. The corresponding AST is given in Figure 2.14 (assuming $\times$ has higher precedence than $+$).

The tree format is easy to traverse, and we can easily change a part of the program by replacing a subtree. From the AST it is immediately clear that there is value multiplied by the constant 2, which is equivalent to a bitwise left shift. The $\times$ node could therefore be replaced with a shift operation without affecting the result of the expression. To make this change to the example expression, the right subtree of $+$ is replaced, as shown in Figure 2.15.

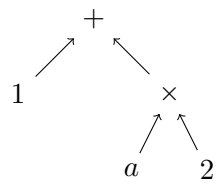


Figure 2.14: Abstract syntax tree for the expression $1 + a \times 2$.

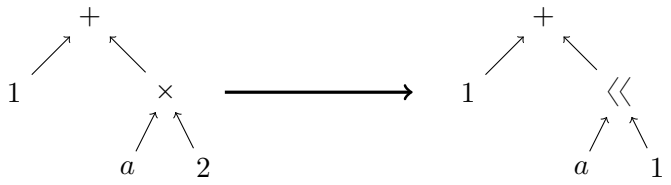


Figure 2.15: Replacing $\times 2$ with $\ll 1$.

2.4.2 Static Single Assignment

Static single assignment (SSA) is a common feature of compiler IRs. A program is in SSA form if all variables are assigned a value upon definition and are not redefined. It is a requirement for using MLIR (see Subsection 2.4.3), so we adhere to it in our compiler. SSA form may seem restrictive, but any program that does not adhere to SSA form can be converted into an equivalent one that does. For example, consider Listing 2.5, which is not in SSA form because `a` is assigned twice. This program is easily converted to SSA form by introducing an extra variable to hold the updated value of `a` and updating all uses of `a` after the reassignment to use the new value.

Listing 2.5: Non-SSA program

```
int a = 42;
a = a + 1;
use(a);
```

Listing 2.6: SSA program

```
int a0 = 42;
int a1 = a0 + 1;
use(a1);
```

Enforcing SSA form in the IR makes many common program analyses and transformations easy to perform. As an example, we will perform constant folding and dead code elimination on Listing 2.6.

1. `a0` is assigned a constant value, and SSA form guarantees this will be the value of `a0` throughout the program. We therefore safely replace all occurrences of `a0` with the value `42`.

2. `a1` now adds two constants, so we compute the result and make `a1` the constant 43.

3. `a1` is now a constant, so all uses can be replaced with its constant value.

4. Both `a0` and `a1` have no remaining uses, so they can be removed. The final program is just the statement `use(43);`

Even conditional assignment can be modeled in SSA, though it requires the addition of special constructs to the IR. The approach we use in our work is *block arguments*. Any value used inside an *if* or *for* is explicitly passed as an argument, and any updated values are returned as outputs. We show another non-SSA program and its conversion to SSA form in Listing 2.7 and Listing 2.8 respectively.

Listing 2.7: Non-SSA program

```
int b = ...;
int c = ...;
int d = ...;
int a;
if (b) {
    a = c;
} else {
    a = d;
}
use(a);
```

Listing 2.8: SSA program

```
int b = ...;
int c = ...;
int d = ...;
int a = if(b) (c, d) {
    block(int arg0, int arg1){
        yield arg0;
    }
} else {
    block(int arg0, int arg1){
        yield arg1;
    }
}
use(a);
```

Variables `c` and `d` are used in one of the branches, so they are passed as arguments to **`if`**. Each branch becomes a block with two arguments. The final `yield` statement sets the value returned from the branch.

This concludes our treatment of SSA in the abstract. In the next section, we discuss MLIR, an IR framework that enforces the SSA form.

2.4.3 MLIR

Multi-Level Intermediate Representation (MLIR) is a collection of tools for the definition, analysis and transformation of intermediate representations. By defining the IR for our DSL in MLIR, we can dramatically reduce the amount of effort required to implement a compiler:

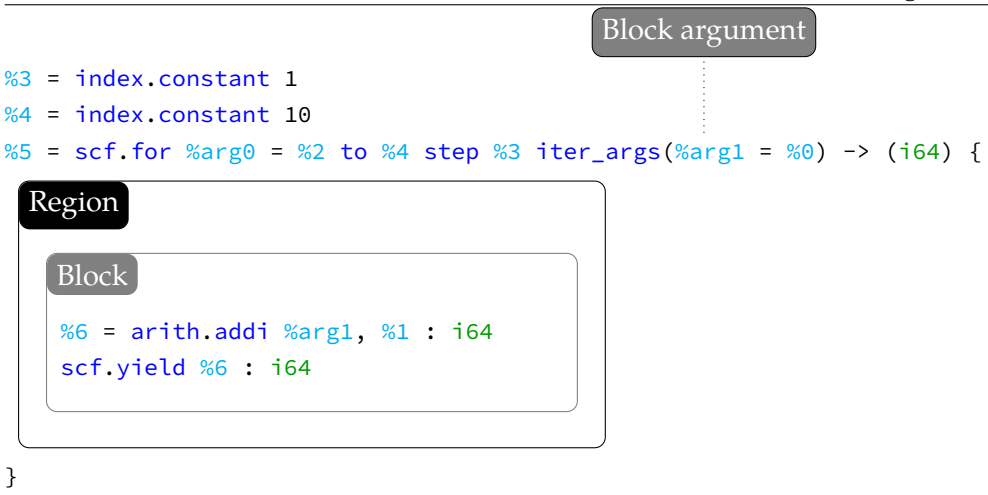
- MLIR provides the Operation Definition Specification (ODS) to declaratively define operations, attributes and types, significantly reducing the amount of boilerplate C++ code.
- IRs defined in MLIR have a standardized textual format. A parser and printer for the IR are automatically generated, with syntax highlighting support in popular editors.
- Commonly-used building blocks for IRs are included with MLIR. All scalar types we need are included in MLIR, including operations such as addition, multiplication, etc. MLIR even offers a tensor type we can use to define matrices and vectors, as well as structural control flow operations (if, for).

MLIR organizes definitions into *dialects*. There is the *builtin* dialect with core types like integers, *arith* for arithmetic operations on scalars, and *scf* for structured control flow operations. Operations can be grouped into a *block*, like statements executed one after the other. A collection of blocks in turn can be put into a *region*, with the last operation in each block determining what block to execute next.

A key feature of MLIR is that it allows operations to have nested regions, making it possible to implement complex constructs like loops as regular operations. The nested structure of MLIR is highlighted in Listing 2.9.

Listing 2.9: A loop in MLIR.

```
%0 = arith.constant 0 : i64
%1 = arith.constant 1 : i64
%2 = index.constant 0
```



The example above also highlights MLIR's use of block arguments, as discussed in Subsection 2.4.2. The design of MLIR is flexible enough to conveniently encode many IR constructs, yet regular enough that we can easily mix operations from different dialects in a single module. This lets us leverage the many existing operations bundled with MLIR, and add only those operations that are specific to our work.

MLIR also provides a standardized infrastructure to analyze and modify programs, which is the topic of the next background section.

2.5 Program Transformation

By definition, compilers translate programs from one language to another. This translation requires that the compiler transforms the operations that make up the input program into new operations that are valid in the output language. Program transformation is therefore fundamental to the implementation of any compiler.

When transforming a program, the compiler must ensure that the behavior of the program remains the same. In other words, all transformations applied must preserve the *semantics* or *meaning* of the program.

We cover the following aspects of program transformation:

- Rewrite rules: a modular approach to transformation based on simple rules that can be combined to produce complex transformation. Small individual rules make it easier to guarantee the overall transformation is semantics-preserving (Subsection 2.5.1).
- Canonicalization: simplification of programs and the removal of redundant constructs. This yields programs that do not perform unnecessary operations and reduces the set of constructs that need to be covered by other rewrite rules (Subsection 2.5.2).
- Optimization: Finding transformations of the program that improve performance (Subsection 2.5.3).
- Lowering: Transforming programs into a different representation that is less abstract and closer to the execution platform (Subsection 2.5.4).

Since rewrite rules are the fundamental building block for canonicalization, optimization, and lowering, we start there.

2.5.1 Rewrite Rules

The *rewrite rule* is a powerful pattern to define transformations using a divide-and-conquer strategy. Rewrite rules define small, local substitutions to be applied to program fragments, and are accompanied by rules that define when they can be applied.

As an example of a rewrite rule, consider that transposing a matrix A twice produces the original matrix:

$$(A^T)^T = A$$

The define rewrite rules, we will use the following syntax:

`<pattern to match>` $\Rightarrow$ `<substitute for match>` **if** `<constraints (optional)>`

Based on this property of the transpose operator, we can define the following rewrite rule:

$$(x^T)^T \Rightarrow x$$

Where x is a placeholder for an arbitrary expression. This rewrite rule can be applied exhaustively to the program by searching for occurrences of the pattern. For example:

$$\begin{aligned} & (((A^T)^T)^T)^T \\ \Rightarrow \quad & ((A^T)^T) \\ \Rightarrow \quad & A \end{aligned}$$

This particular rewrite rule can be safely applied exhaustively to a program. Every rewrite makes the program smaller, so there must necessarily be a point at which the program cannot be reduced any further. Not all rewrite rules have this convenient property. Consider for example the commutativity of addition, which we express with the following rewrite rule:

$$x + y \Rightarrow y + x$$

Applying this exhaustively to the following example program, the rewriting process does not terminate:

$$\begin{aligned} & 1 + a \\ \Rightarrow \quad & a + 1 \\ \Rightarrow \quad & 1 + a \\ & \dots \end{aligned}$$

As we will see in Subsection 2.5.3, non-terminating rewrite rules are still useful in the compiler context, so long as the compiler places a limit on the number of rewrites performed.

Compilers typically combine a large set of rewrite rules. Though each of these rules only makes a small change to the program, by applying many different rules repeatedly, programs can be transformed dramatically. The key advantage of individual rewrite rules is that to prove the correctness of the overall transformation, we only have to prove the correctness of the individual rewrite rules, which are small and limited in scope.

In the next sections, we discuss various applications of rewrite rules in compilers.

2.5.2 Canonicalization

Canonicalization is the simplification of programs by removing redundant constructs. We use the term canonicalization to be consistent with MLIR, in other literature it is often referred to as *normalization*. We have already seen one example of redundancy in Subsection 2.5.1, where two transpose operations cancel each other. Another example of redundancy is the calculation of values that are not needed. Consider the program in Listing 2.10.

Listing 2.10: A program with unnecessary computation

```
int a = ...;
int b = computeB();
print(a);
exit(0);
```

in the program above, a function `computeB` is invoked to produce the value for variable `b`, but this value is not used anywhere in the program. If `computeB` is known not to have any side effects, a compiler can omit the call entirely. Known as *dead code elimination*, this pattern may be expressed with the following rewrite rule:

$$v = e \quad \Rightarrow \quad \mathit{nil} \quad \textbf{if} \quad \neg\text{used}(v) \wedge \neg\text{haveSideEffect}(e)$$

In Listing 2.11, we give a program that demonstrates yet another form of redundancy.

Listing 2.11: A program that computes the same value twice

```
int a = ...;
int b = ...;
int c = a + b;
int d = a + b;
print(c+d);
```

Variables `c` and `d` hold the same value, but they are computed independently. We say that `a+b` is a *common subexpression*. The transformation that removes the redundancy is called *common subexpression elimination* (*CSE*). It is implemented by searching for an already defined variable that is assigned the same value as the target expression. The rewrite rule is defined as:

$$e \Rightarrow v \quad \textbf{if} \quad (v, e) \in \mathcal{V}$$

Where $\mathcal{V}$ is the set of already defined variables and their expressions.

Canonicalization rules can always be safely applied exhaustively since every rewrite leads to a simpler version of the program, which makes it easy and cheap to apply. Our compiler uses canonicalization after every transformation phase to clean up redundant code introduced by other rewrites. In the next section, we cover rules that need to be applied more carefully.

2.5.3 Optimization

Optimization is the modification of programs to make them more efficient while preserving their behavior. In Section 2.2 we covered query planning, which is a special case of program optimization where the program is a query. To support complex optimizations while maintaining correctness, optimization too relies on rewrite rules. The example rewrite rules seen in Subsection 2.5.2 both improve the efficiency of the final program, and can therefore be classified as optimizations. For these rules, it is immediately obvious that they are beneficial, so they can be applied liberally. A good optimizer however cannot rely on canonicalization rules alone. In some cases, a more thorough exploration of the program space is necessary to find a more efficient version of the program. Such an exploration may require multiple rewrite rules to be applied, and not all of them may immediately seem beneficial.

As an example, consider the expression $(a - 1) \cdot (a + 1)$. This expression contains 3 arithmetic operations, but a good optimizer can reduce this to 2 by using the distributivity of addition:

$$
\begin{aligned}
& (a-1)\cdot(a+1) && \text{(3 operations)} \\
\Rightarrow\quad & a^2+a-a-1 && \text{(4 operations)} \\
\Rightarrow\quad & a^2-1 && \text{(2 operations)}
\end{aligned}
$$

Initially, when the distributivity rewrite is applied, the number of operations grows to 4, but it is necessary to allow the second rewrite to fire, which brings the number of operations down again. The distributivity rewrite is only beneficial here because of the special relationship between the operands, allowing the a variables to cancel out. It is therefore not a canonicalization rule, but it can be part of the ruleset for an optimizer.

For another example, we look at *join ordering*, a classic query optimization problem[49]. A property of the $\bowtie$ (join) operator (see Subsection 2.2.1) is that it is *associative*. This means that we can define the following rewrite rule:

$$(x \bowtie y) \bowtie z \Rightarrow x \bowtie (y \bowtie z)$$

This rewrite rule, like the distributivity rule, should not be applied as a canonicalization rule, because it does not always improve the query plan. Instead, the classical optimization strategy is to enumerate all possible orderings of joins, estimate their runtime cost, and pick the ordering with the lowest cost. Provided that the cost estimate is accurate enough, a full enumeration is guaranteed to produce the optimal query plan.

After a program has been optimized, there is one more transformation left, which we discuss in the next section.

2.5.4 Lowering

At its most abstract level, *lowering* is a transformation step that converts a program expressed in a high-level, abstract IR, into a lower-level IR that is closer to the language natively supported by the execution platform. We have already seen one example of this in the conversion from logical into physical plans (see Section 2.2). Another example is LLVM [28], where after optimization the LLVM IR is lowered to native machine code.

Lowering often comes down to finding a sequence of simple low-level instructions that together implement a complex high-level operation. Take for example the matrix multiplication $A \cdot B$ as a high-level operation. It can be lowered to relational algebra by combining a join, apply and aggregation operator, as shown in Figure 2.16. This specific lowering is covered in detail in Chapter 6.

In other cases, the high-level IR may have one operation for a group of specialized functions that exist at the low level, and we must select one. An example of this is the selection of a particular join implementation, as seen in Section 2.2.

A program may go through successive stages of lowering before it reaches its final form, which can be machine code or a representation that can be executed by an interpreter. In Chapter 5 we will see a pipeline that performs multiple lowering steps, with additional optimization between each lowering.

This concludes our treatment of background topics on databases and compilers. The next chapters describe the design of the language (Chapter 3), the compiler architecture (Chapter 4) and the backends (Chapter 5 and Chapter 6).

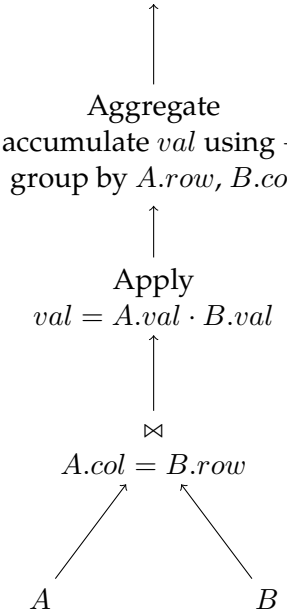


Figure 2.16: Matrix multiplication $A \cdot B$ lowered to relational algebra

Chapter 3

Language Design

This chapter discusses the design of the graph algorithm DSL, *graphalg*. We start by stating the design goals of graphalg in Section 3.1. In Section 3.2 we describe the established computational models for the implementation of graph algorithms, and argue that linear algebra is the best fit for our goals. Finally, Section 3.3 provides a detailed overview of the resulting language.

3.1 Design Goals

The graphalg language is designed to be an accessible interface to developing and running graph algorithms on large graphs stored in AvantGraph. This use case motivates the following goals for the language:

- Efficient: AvantGraph can store graphs with billions of vertices and edges. Graphalg is designed to be efficiently executed by the different backends, with high data parallelism and low memory usage.
- Ease of programming: We do not assume the users of graphalg to be expert programmers. We aim to provide an interface that is familiar to engineers of many different backgrounds and handles parallelization and memory management automatically.
- Safe to run inside a database: Databases are typically shared by many users. Individual users must not crash the database or consume excessive resources.
- Expressive: Graphalg should be flexible enough that many different types of algorithms can be written in it.

We now discuss each of these goals in more detail.

3.1.1 Efficient

Users of graphalg will run queries on very large graphs and expect answers fast. To improve performance, the implementation must be multi-threaded to make use of all available CPU cores. It must also avoid copying large datasets as much as possible since large graphs may not fit into main memory (and AvantGraph has no facilities to spill intermediate results to disk). Finally, since graphalg users may not be expert programmers, they may not write the most efficient implementation of an algorithm. To compensate for this to some degree, the system should be able to perform aggressive optimization of programs. AvantGraph already performs such optimizations on query plans by using statistics known about the data. The same optimizations should be performed on graphalg programs.

All of these requirements apply chiefly to the runtime, but we also take them into account when designing the language to ensure that generating fast code is feasible.

3.1.2 Ease of Programming

While we want to support efficient implementations of algorithms, we cannot shift the burden of writing highly optimized code onto our users, who are not guaranteed to be expert programmers. Firstly, we choose not to include explicit parallelism in the language. While explicit parallelism is sometimes needed to achieve high performance, it requires users to understand locks and atomic operations, and that they avoid concurrency bugs such as race conditions. On top of this, atomic operations are also notoriously difficult for compilers to optimize [33].

Secondly, graphalg should have automatic memory management. We expect users to be most familiar with languages that have automatic memory management like Python and MATLAB, so graphalg should offer a similar interface. An additional important benefit of automatic memory management is that it prevents leaking memory, which also contributes to the next requirement.

3.1.3 Safe to run inside a database

Running user-provided code inside a system shared by many users is potentially dangerous. If graphalg programs are run unsandboxed and with full access to the memory, they may interfere with the queries of other users. A program could for example access memory that belongs to another query and corrupt its results. Alternatively, it could overwrite the internal data structures of the database and cause it to misbehave or crash. To avoid programs interfering with other queries or the database while avoiding the runtime overhead of sandboxing, we require graphalg to be memory safe.

Another risk is that programs do not terminate, and continue to hog resources until they are forcefully terminated. Non-terminating behavior is confusing to users, wastes computational resources, and may prevent other users from running queries, so we require that graphalg programs are terminating. Proving termination of a program written in a Turing-complete language is undecidable, so this requirement implies graphalg can not be Turing-complete. However, as we will see in the later chapters this is not a severe limitation for expressivity in practice.

3.1.4 Expressive

We have already defined requirements for the language to be efficient, easy to program and safe. All of these can be trivially satisfied by a language so limited in functionality that no useful programs can be written in it. For this reason, we define a notion of expressivity for graphalg. Graphalg should be flexible enough that it can be used to implement a wide variety of graph algorithms. Rather than a complex formal definition of the required expressivity, we select a sample of different algorithms that graphalg must be able to encode. Graphalg is defined by generic operations that are not tailored to any specific algorithm, so if it is expressive enough to accommodate the algorithm in the sample, likely, it will also be able to encode many other algorithms. The algorithms we use to evaluate the expressiveness of graphalg are the programs in the GAP benchmark suite [3], which is also used to evaluate the performance of the system:

- Breadth-First Search (BFS) traverses the graph starting from a given source vertex. BFS first visits all vertices at the current depth (the number of edges to the source vertex), before visiting nodes at a greater depth. It labels each vertex in the graph with its 'parent node', the vertex through which they were first discovered.

- Single-Source Shortest Paths (SSSP) computes the distances of all vertices in the graph to a given source vertex.

- PageRank (PR) computes the PageRank score for all vertices.
- Connected Components (CC) divides a graph into its connected components and labels each vertex with its connected component id.
- Betweenness Centrality (BC) approximates the betweenness centrality of the vertices in the graph by computing shortest paths from a subset of the vertices.
- Triangle Counting (TC) counts the number of triangles (three vertices connected in a loop) in the graph.

3.2 Computational Models Considered

The key decision in the design of our language is choice of computational model. In this section, we explore four options that are popular in the literature as well as in industry.

3.2.1 General-Purpose Imperative

Arguably the simplest approach would be to embed an existing programming language into the database. For example, Neo4J allows writing user-defined procedures in Java[53]. Another example is UDO[48], an implementation of user-defined operators for Umbra, where the user implements a custom query operator in C++. Having the full functionality of a general-purpose language available means those systems are highly expressive. UDO also manages to achieve very high performance, but the user is responsible for managing memory and parallel execution. Since UDO operators are written in C++ and compiled together with the rest of the query plan, bugs can lead to memory corruption anywhere in the address space of the database. By being written in a memory-safe language, Neo4J stored procedures do not have the same issue, but they still do not guarantee termination. We consider Neo4J's approach to be easier to use, but it is less efficient than UDO.

In summary, general-purpose languages do not meet our design goals on safety, and struggle to combine efficiency and ease of programming.

3.2.2 Vertex-Centric Processing

A popular approach in distributed graph processing framework, the vertex-centric model encourages its users to 'think like a vertex' [36]. Algorithms are implemented as programs that run on every vertex concurrently. They can each update a property belonging to their vertex or one of their outgoing edges, and communicate with other vertices by sending messages. The approach was popularized by Google Pregel [34] but is now implemented by many, mostly distributed, graph processing frameworks [36]. While the scalability of the vertex-centric approach to large clusters is often touted, it is known to suffer from efficiency issues [26], particularly compared to shared-memory systems [37].

Since efficiency is an important design goal, and AvantGraph is a single-node system, we do not consider the vertex-centric model suitable for our use case.

3.2.3 Edge and Vertex Sets

The edge and vertex sets model is similar to a standard imperative model, except that operations on the graph structure are always performed in bulk, over sets of either vertices or edges. The model strikes a careful balance between being expressive (by having imperative control structures), yet amenable to analysis and optimization (by operating on the graph with high-level bulk operations). Languages such as Greenmarl [23] (now Oracle PGX [7]) and GraphIt [60] that implement this model have shown excellent performance on graph

analytics benchmarks [2]. The edge and vertex sets model is efficient and expressive and, as Oracle's production deployment of PGX demonstrates, it can be safely integrated into a database. PGX and GraphIt also have automatic memory management, but they do require the user to think about parallelism in their code. PGX models concurrent access with special variables that support only aggregation operations, whereas GraphIt has explicit atomic accesses. This appears to be an inherent problem with the edge and vertex sets model: blocks of user code are running in parallel for multiple vertices/edges, and need some way to communicate.

We consider the edge and vertex sets model a strong candidate with a good track record in efficiency. However, we find that languages based on it still have explicit parallelism, which we would like to avoid.

3.2.4 Linear Algebra

Our choice of computational model is linear algebra. The use of linear algebra for graph analytics has been extensively studied [25]. On the implementation side, there is GraphBLAS [12], an open standard for sparse linear algebra primitives designed specifically for graph analytics. Many important graph algorithms have a direct analog in linear algebra or can be adapted to it [50, 59, 35].

In terms of efficiency, the reference GraphBLAS implementation SuiteSparse has shown good performance on the GAP benchmark suite [2], though not as good as e.g. GraphIt. Linear algebra operations are higher-level than those in the other models we have considered and have a solid theoretical basis, which makes them highly amenable to analysis and optimization.

We also consider the use of linear algebra to be beneficial for the accessibility of the language. Linear algebra is part of the curriculum for many engineering degrees, so we can expect that many users will be familiar with it, even if they do not have much programming experience. Looking at the Python and Julia bindings for GraphBLAS [42], we can see that graph algorithms can be implemented in this model with automatic memory management and without any explicit parallelism. Computationally expensive operations like matrix multiplication are executed on multiple threads, but this parallelism is invisible to the programmer.

By using linear algebra as a basis for our DSL, we can also make it safe to run inside a database. All data access is performed on scalar values and matrix types. As long as we perform bounds checking on matrix element access, programs cannot corrupt memory that does not belong to them. Our language is built on top of basic linear algebra operations such as element-wise products and matrix multiplication which are known to be terminating. By appropriately restricting the control structures in our DSL (bounding loops and preventing recursion), we can also guarantee that all programs are terminating.

Finally, we know that a linear algebra-based language can be expressive enough to implement all programs from the GAP benchmark suite, because the LAGraph [35] library provides reference implementations of all GAP benchmark programs built on top of GraphBLAS.

In conclusion, we find that linear algebra satisfies all our design goals. While not as efficient as some of the other candidates, it still offers good performance and exceptional opportunities for optimization. It effectively hides parallelism and memory management, offering a simple programming interface. Linear algebra is also a familiar tool to many engineers, which we believe makes the language accessible to a wider audience.

With the choice of the computational model out of the way, we discuss the other details of the language in the next section.

3.3 Language Reference

In this section, we give a comprehensive overview of graphalg and explain key design decisions. We start with the fundamental constructs present in any language, such as data types, expressions and control flow. Then we cover the linear algebra operations that are specific to graphalg and discuss the integration of the language into AvantGraph queries.

3.3.1 Data Types

Graphalg supports the following primitive types:

- `bool`: Booleans (`true`, `false`)
- `int`: Integers (64-bit, signed)
- `real`: Floating point numbers (64-bit IEEE 754)
- `index`: Index type / Vertex ID (64-bit unsigned)

This is the largest set of types supported by both backend runtimes (AvantGraph and GraphBLAS), and it is sufficient to encode all programs we use in the evaluation. Booleans are not strictly necessary since they can be emulated using integers, but the smaller boolean type is more memory efficient, which we also found to have a measurable effect on execution time.

Graphalg has two generic types, `Vector<T>` and `Matrix<T>`, which are defined for T $\in$ {`bool`, `int`, `real`} (`index` is not included). `Vector<T>` and `Matrix<T>` represent sparse matrices: they have a defined size, but not all entries may have a value. An entry that is not present is distinct from an entry set to the zero value. The following two matrices are not equivalent to each other:

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \neq \begin{bmatrix} & 1 \\ 2 & 3 \end{bmatrix}$$

`Vector<T>` is a specialization of `Matrix<T>` where the number of columns is one (a column vector).

Primitive types are instantiated directly with their value, while vector and matrix types are instantiated as empty matrices:

Listing 3.1: Instantiating different types in graphalg

```
b = true;
i = 42;
r = 2.14;
idx = index(42);
v = Vector<int>(10);
m = Matrix<int>(10, 10);
// Vector and Matrix can be dynamically sized
m2 = Matrix<int>(i, i);
```

To assign values to the entries of a matrix, assignment syntax is used:

Listing 3.2: Assigning to entries in the matrix

```
v = Vector<int>(10, 10);
// assign one element
v[0] = 42;
// assign to a range
```

```
v[0:5] = 42;
// assign all elements
v[:] = 42;
```

The bounded range written as `[a:b]` denotes the half-open interval $[a, b)$, (start inclusive, end exclusive).

Instances of `Vector<T>` and `Matrix<T>` expose additional properties to read their dimensions (`nrows`, `ncols`) and number of present entries (`nvals`):

Listing 3.3: Matrix properties

```
v = Vector<int>(10);
v.nrows // -> 10
v[0] = 1;
v[1] = 2;
v.nvals // -> 2
m = Matrix<int>(10, 20);
// only available on Matrix<T>
m.ncols // -> 20
```

3.3.2 Functions

Functions define a section of code with zero or more typed parameters and a single return value. All graphalg code must be defined inside of a function. A single graphalg program may consist of multiple functions that can refer to each other provided the call graph is acyclic. Keeping the call graph acyclic prevents (mutual) recursion, which can cause non-terminating behavior. Another advantage of an acyclic call graph is that any function call can be safely inlined. This is an important property for the operators backend, which has no notion of functions.

Listing 3.4: Examples of functions.

```
func Add(a:int, b:int) -> int {
      return a + b;
}

// This pair of mutually recursive functions are not allowed
func RecursiveA() -> int {
      return RecursiveB();
}
func RecursiveB() -> int {
      // Creates a cycle in the call graph -> compiler error
      return RecursiveA();
}
```

3.3.3 Expressions

Graphalg supports the usual arithmetic operations $+, -, *, /$ over integers and real numbers, respecting the typical rules of precedence (multiplication and division have higher precedence than addition and subtraction). The inputs to operators must have the same type. To apply an operator to values of different types, one of them must be cast to a matching type using one of the cast operators `int(·)`, `real(·)` or `index(·)`.

Listing 3.5: Examples of arithmetic.

```
a * b + c * d;              // is equivalent to (a * b) + (c * d)
```

```
intVal = 1 + int(2.14);       // -> 3
realVal = 1.23 + real(4);     // -> 5.23
```

Comparison operators (`=`, `!=`, `<`, `<=`, `>`, `>=`) are also available to compare scalar values of the same type.

3.3.4 Variables and Scoping

Expressions can be assigned to variables to allow their value to be used in one or more later expressions. Variables are accessible from within the scope they are defined, and from any scopes nested inside the definition scope.

While all values in graphalg are immutable, variables may be reassigned to new values. Like primitive values, `Vector<T>` and `Matrix<T>` instances have value semantics. We do not presume value semantics to be objectively better than the reference semantics common in other languages. Rather, we choose value semantics for graphalg because the operators backend (see Chapter 6) cannot support in-place updates. MLIR's built-in `tensor` type that we use internally to represent matrices also has value semantics, allowing us to reuse the standard MLIR infrastructure as much as possible.

Assigning to the entries of a matrix creates a new matrix and updates the variable to point to it. The program below illustrates this:

Listing 3.6: Example of value semantics

```
a = Vector<int>(2);
// b points to the same value as a.
b = a;
// variable a is reassigned to a new vector filled with 42,
// while variable b still points to an empty vector.
a[:] = 42;
// a -> [42;42]
// b -> [_;_]
```

This differs from a language like Python where matrices have reference semantics. A similar program in Python would produce a different output:

Listing 3.7: Reference semantics in Python

```
import numpy as np
a = np.array([0, 0])
b = a
# updates the vector in-place
a[:] = 42
print(a) # -> [42 42]
# b also sees updates to a
print(b) # -> [42 42]
```

The GraphBLAS library does have reference semantics, so in the GraphBLAS backend (Chapter 5) there is an additional step in the compilation phase which transforms operations with value semantics into operations with reference semantics.

3.3.5 Control Flow

Besides function calls, graphalg supports two types of control flow: conditional branching, and bounded loops. The `if` statement runs one of two branches depending on the value of the condition. All variables from the outer scopes are visible within an if branch and can

be reassigned there. There are no operations in graphalg with side effects, so updating a variable is the only way for an `if` statement to influence later program behavior.

Listing 3.8: A simple if statement

```
// define a in the outer scope to allow accessing it
// outside of the if branches.
a = 0
if c > 0 {
      a = 42;
} else {
      a = 43;
}
// variable a holds 42 or 43, depending on the value of c
```

The other control flow construct is the `for` loop. It executes its loop body for zero or more iterations, depending on the provided range. A for loop creates an additional variable, visible only within the loop body, called the *iteration variable*, that stores the current value within the iteration range We show an example of a loop below:

Listing 3.9: A function containing a simple for loop

```
func Fibonacci(n:index) -> int {
      a = 0;
      b = 0;

      for i in 0:n {
            // iteration variable 'i' available in this scope
            // loop body executes once for every value of i in interval [0,n)
            t = a + b;
            b = a;
            a = t;
      }

      return a;
}
```

An optional break condition may be added to terminate the loop early before the end of the iteration range is reached. It is checked before each iteration, terminating the loop early if it evaluates to `true`.

Listing 3.10: A loop with a break condition

```
for i in 0:maxIterations until converged {
      // update value in a loop until it converges,
      // or until we reach the maximum number of iterations.
      newValue = step(value);
      // have we converged yet?
      converged = newValue == value;
      value = newValue;
}
```

Both endpoints of the iteration range are required. This is the second part of graphalg's termination guarantee: all loops are bounded, so they are guaranteed to terminate.

3.3.6 Linear Algebra operations

Linear Algebra operations are a distinguishing feature of the language. We support a large subset of the GraphBLAS API as native operators. Many programs written in e.g. C using GraphBLAS library calls have a direct mapping to graphalg. Moreover, because these operations are built-in to graphalg, the resulting program is more concise and syntactically clearer.

Matrix Multiplication

To multiply two matrices `A` and `B`, write `C = A (+.*) B`. This computes the entries of `C` as:

$$C_{ij} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \cdots + a_{in} + b_{nj}$$

Matrix multiplication is a useful primitive for implementing graph algorithms. As an example, consider the program given in Listing 3.11, which computes the number of distinct paths from a given vertices to every other vertex in the graph. To demonstrate that matrix multiplication indeed relates to computing distinct paths in a graph, we illustrate the execution of the program on an example graph in Figure 3.1.

Listing 3.11: Number of distinct paths from source to all vertices in the graph

```
func NPaths(graph: Matrix<int>, source:index) -> Vector<int> {
   paths = Vector<int>(graph.nrows);
   front = Vector<int>(graph.nrows);
   front[source] = 1;

   for k in 0:graph.nrows {
       paths += front;
       // Vector-Matrix multiply
       front = front (+.*) graph;
   }
   return paths;
}
```

In Listing 3.11, the matrix multiplication represents a graph traversal step. By running the multiplication in a loop the entire graph can be explored.

Graphalg allows the user to specify different operators from matrix multiplication than the standard addition and multiplication to change the graph property to compute during the traversal. For example, we can write a very similar program to compute the shortest paths in a graph, shown in Listing 3.12.

Listing 3.12: Single source shortest path algorithm in graphalg.

```
func SSSP(graph: Matrix<int>, source: index) -> Vector<int> {
    v = Vector<int>(graph.nrows);
    v[source] = 0;

    for i in 0:graph.nrows {
        v min= v (min.+) graph;
    }

    return v;
}
```

Instead of multiplying the number of paths and adding them, we sum the path lengths and select the minimum length, which gives the shortest path from the source vertex to every

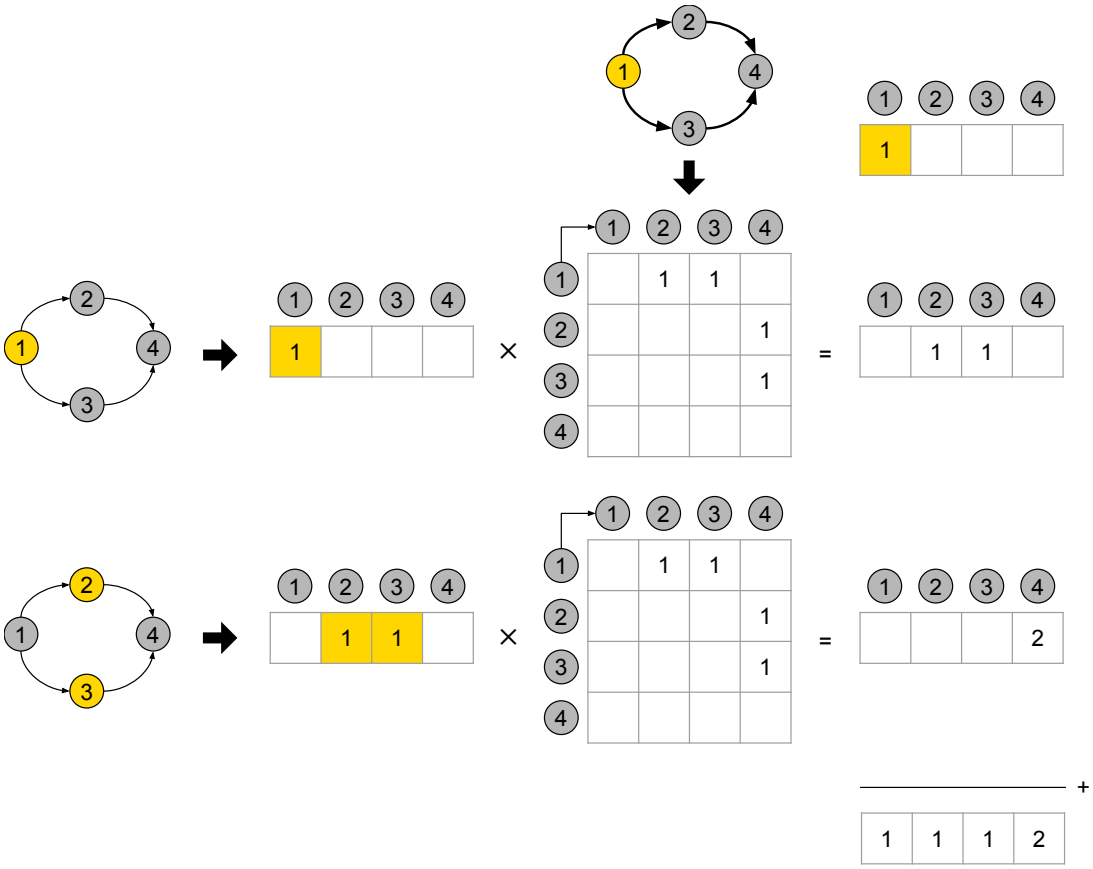


Figure 3.1: Visual representation of the execution of the `NPaths` algorithm given in Listing 3.11.

other vertex in the graph. The combination of an addition monoid and a binary operator for multiplication that together define the behavior of generalized matrix multiplication is referred to as a *semiring*[25]. The particular semiring `(min.+)` used to compute shortest paths is also known as the *tropical semiring*. A full list of supported monoids and binary operators that can be combined into a semiring is given in Section 3.3.6.

Since `Vector<T>` always stores a column vector, it must be transposed before it can be used on the left-hand side of a matrix multiplication. Graphalg adopts the same convention as GraphBLAS, and performs this transpose explicitly.

Element-wise operations

To apply a binary function element-wise to the entries of two matrices, graphalg provides two operators:

Listing 3.13: element-wise operations

```
C_intersect = A .+ B;
C_union = add(+, A, B);
```

If all entries of `A` and `B` are present, both operations produce the same output. The difference is in how missing entries in one of the two matrices are handled. The set intersection variant $C = A.\oplus B$ computes C as:

$$C_{ij} = A_{ij} \oplus B_{ij} \quad \text{where} (i,j) \in A \cap B$$

Only entries that are present in both A and B will produce an entry in the output C. Alternatively, the `add` operator can used, which also copies entries that are only present in one of the two matrices:

$$
\begin{aligned}
C_{ij} &= A_{ij} \oplus B_{ij} && \text{where} (i,j) \in A \cap B \\
C_{ij} &= A_{ij} && \text{where} (i,j) \in A \backslash B \\
C_{ij} &= B_{ij} && \text{where} (i,j) \in B \backslash A
\end{aligned}
$$

Apply

When the function to be applied to entries is a unary operator, or if one of the operands is a scalar value, the `apply` operator may is used instead.

Listing 3.14: Example of the `apply` operator

```
// Absolute value
C = apply(abs, A);

// Increment all entries by one
C = apply(+, A, 1);
```

See Table 3.3 and Table 3.4 for the list of supported operators.

Select

The `select` operator applies a predicate to each entry of the input matrix and outputs a new matrix with the elements for which the predicate holds. Predicates can be based on the value of the entry or its position in the matrix:

Listing 3.15: Example of the `select` operator

```
// Keep only non-zero elements
C = select(!=, A, 0);

// Keep only the entries located below the zeroth diagonal.
// the last input `-1` sets the maximum diagonal.
C = select(tril, A, -1);
```

Supported predicates are listed in Table 3.5.

Reduce

`reduce` accumulates all entries of a matrix into a single scalar value, merging values using the given monoid (see Table 3.2). for `Matrix<T>` instances, entries can also be accumulated row-wise using the `reduceRows` operator. As an example, the program below computes the total number of edges in the graph, and the out-degree of each vertex.

Listing 3.16: Example of the `reduce` and `reduceRows` operators

```
// create a matrix with value '1' for each entry present in A
// operator 'one' is defined as one(x) = 1
ones = apply(one, A);

// note: real programs should use A.nvals
nvals = reduce(+, ones);
```

```
// number of outgoing edges per vertex.
outDegree = reduceRows(+, ones);
```

Accumulate

The result of an expression may be accumulated into an existing matrix rather than replacing it. To enable accumulation, the monoid to use should be specified before the `=` symbol, as shown in Listing 3.17. See Table 3.2 for a list of supported monoids.

Listing 3.17: Example of accumulation

```
// Define C
C = ...;

// Variable C assigned to value of A.
// The previous value of C is lost.
C = A;

// Original value of C and the value of B are added together.
// C points to the resulting new value.
C += B;

// The previous two statements are equivalent to
// using add.
C = add(+, A, B);
```

As we show in the example, the accumulation syntax is equivalent to using `add`. Even though it is therefore technically redundant, we choose to keep this syntax for two reasons:

- Multiply and accumulate loops are very common in graph algorithms. Offering a dedicated accumulation syntax makes this pattern easy to identify visually.
- Accumulation syntax provides a hint to the compiler that the accumulation may be performed in place. The GraphBLAS backend can typically optimize accumulation syntax into an in-place operation, whereas if `add` is used the result is always stored in a newly allocated buffer.

Masked assignment

When assigning a matrix to a variable, it is possible to specify an additional mask matrix to control which elements to copy. The syntax for this is `C<M> = A`. Values of `A` are copied to `C` if `M` has value `true` at that index. If the value of `M` is `false` or there is no entry present, the existing value of `C` (if any) is preserved. The new value of `C` is then defined as:

$$
\begin{aligned}
C_{ij} &= A_{ij} && \text{if} \quad M_{ij} = \texttt{true} \\
C_{ij} &= C_{ij} && \text{if} \quad (i,j) \notin C_{ij}
\end{aligned}
$$

The following additional flags may be provided to change the details of the operation:

- The mask can be complemented by prefixing it with an exclamation mark. `C<!M> = A` copies all values from `A` for which $M_{ij} =$ `true` does *not* hold. Note that this does not just copy values from `A` where the mask entry is `false`, but also entries that are not present in `M`.

- `struct` makes the mask structural, changing the copy condition from $M_{ij} =$ `true` into $(i, j) \in M$. This is particularly useful if `M` is not a boolean matrix.
- `replace` skips the preservation of existing values in `C`. The final output will contain only the values copied from `A`.

Listing 3.18: Examples of masked assignment.

```
C = Vector<int>(3);
M = Vector<bool>(3);
M[1] = false;
M[2] = true;
A = Vector<int>(3);
A[:] = 1;

C[:] = 0;
// regular mask
C<M> = A;               // -> [0, 0, 1]

C[:] = 0;
// complemented mask
C<!M> = A;              // -> [1, 1, 0]

C[:] = 0;
// structural mask
C<M, struct> = A;       // -> [0, 1, 1]

C[:] = 0;
// replace
C<M, replace> = A;      // -> [_, _, 1]
```

Transpose

The transpose of a `Matrix<T>` can be obtained using the `.T` property. `Vector<T>` always stores a column vector and does *not* support the transpose property.

Listing 3.19: Transpose example

```
// Both branches compute the same result.
// Depending on the in-memory representation,
// one may be faster than the other.
if push {
      C = v (+.*) A;
} else {
      C = A.T (+.*) v;
}
```

Supported operators

Below we list the operators supported as arguments to the linear algebra operations. The current set of operators is sufficient for a wide variety of algorithms, although some may still benefit from custom operators, particulary when combined with user-defined types. Support for custom operators and types is left as future work.

Table 3.2: Monoids

Symbol	Implementation	Notes
+	$x + y$	
-	$x - y$	
max	$max(x, y)$	
min	$min(x, y)$	
any	x	When used as an accumulator, ignores all but the first value

Table 3.3: Unary operators

Symbol	Implementation	Notes
abs	|x|	
one	1	

Table 3.4: Binary operators

Symbol	Implementation	Notes
+	$x + y$	
-	$x - y$	
*	$x \cdot y$	
/	x/y	
==	$x == y$	
!=	$x \neq y$	
<	$x < y$	
<=	$x \leqslant y$	
>	$x > y$	
>=	$x \geqslant y$	
&&	$x \wedge y$	Logical AND
||	$x \vee y$	Logical OR
first	x	
second	y	
secondi	y_{row}	row index of y

3.3.7 Memory management hints

Graphalg supports `swap` and `clear` primitives to provide hints to the compiler for more efficient memory management in backends that lower into buffers with reference semantics. In the GraphBLAS backend use of these hints may change the generated code, if the compiler determines the optimization can be performed without affecting the results. The operators backend does not make use of these hints since it does not perform in-place operations.

`swap` takes two variables as input and swaps their assigned values. Using `swap` suggests the compiler to swap the pointers to the buffers, rather than copying the contents of the buffers. It is functionally equivalent to (but potentially more efficient than) swapping the values through a temporary variable.

`clear` takes one variable as input and assigns to it a new empty matrix of the same size. Using `clear` suggests the compiler to reuse the existing buffer by erasing it. It is equivalent to creating a new empty matrix and assigning it to an existing variable.

Listing 3.20: Examples of `swap` and `clear`.

```
swap A B;
// equivalent to:
T = A;
```

Table 3.5: Predicates

Symbol	Implementation	Notes
tril	$x_{col} \leqslant (x_{row} + y)$	true for entries on or below the yth diagonal
triu	$x_{col} \geqslant (x_{row} + y)$	true for entries on or above the yth diagonal
==	$x == y$	
!=	$x \neq y$	
<	$x < y$	
<=	$x \leqslant y$	
>	$x > y$	
>=	$x \geqslant y$	

```
A = B;
B = T;

clear A;
// equivalent to:
A = Matrix<int>(A.nrows, A.ncols);
```

3.3.8 Grammar

In the previous section we have informally described the operations available in graphalg. To complement the example code presented there, we provide the detailed grammar for the language in Listing 3.21.

Listing 3.21: Grammar describing the syntax of the graphalg language.

```
program   ::= func_def*
func_def  ::= func IDENT [[param_def]]? [[, param_def]]* [[-> type]]? { [[stmt]]* }
param_def ::= IDENT : type
ptype     ::= bool | int | index | real
type      ::= ptype | Matrix<ptype> | Vector<ptype>
stmt      ::= return expr;
            | IDENT [[<mask>]]? [[[ range [[, range]]? ]]]? [[ + | - | min]]? = expr;
            | for IDENT in range [[until expr]]? block
            | if expr block [[else_if]]* [[else block]]?
            | swap IDENT IDENT;
            | clear IDENT;
block     ::= { [[stmt]]* }
else_if   ::= else if expr block
mask      ::= [[!]]? IDENT [[, replace]]? [[, struct]]?
range     ::= [[expr]]? : [[expr]]?
            | expr
expr      ::= ( expr )
            | expr . T
            | expr . nrows
            | expr . ncols
            | expr . nvals
            | expr [ range [[, range]]? ]
            | expr semiring expr
            | expr binop expr
            | expr . binop expr
            | select(binop, expr, expr)
```

```
            | reduce(binop, expr)
            | reduceRows(binop, expr)
            | apply(unop, expr)
            | apply(binop, expr, [[expr]]?)
            | add(binop, expr, expr)
            | - expr
            | ! expr
            | REAL_LITERAL
            | INT_LITERAL
            | true
            | false
            | IDENT ( [[expr]]? [[, expr]]* )
            | IDENT
            | Matrix<ptype>(expr, expr)
            | Vector<ptype>(expr)
            | int(expr)
            | real(expr)
            | index(expr)
monoid   ::= + | - | max | min | any
binop    ::= +
            | -
            | *
            | /
            | ==
            | !=
            | <
            | <=
            | >
            | >=
            | &&
            | ||
            | first
            | second
            | secondi
unop     ::= abs
            | one
semiring ::= monoid . binop
```

3.3.9 Integration with Graph Queries

Graphalg programs can be executed in AvantGraph by embedding them into queries. For our integration, we reuse the existing `CALL` syntax available in OpenCypher. Our only addition to the OpenCypher parser is a new variation of the `WITH` clause. We show an example in Listing 3.22.

Listing 3.22: A Cypher query with an embedded graph algorithm.

```
WITH ALGORITHM "
func TriangleCount(graph: Matrix<bool>) -> int {
    L = select(tril, graph, -1);
    U = select(triu, graph, 1);
    C<L, struct> = L (+.one) U.T;
```

```
    return reduce(+, C);
}
"
CALL TriangleCount()
RETURN count
```

The `WITH ALGORITHM` clause is placed at the top of the query. It takes a string as an argument that contains the graphalg algorithm. After the definition of the algorithm, functions defined inside the algorithm can be invoked using `CALL`.

The OpenCypher syntax does have one important drawback. The graph is always passed as an implicit input to the algorithm, with no way to first preprocess the graph. Unfortunately, none of the supported query languages in AvantGraph provide such functionality. For this thesis, we have chosen to focus on the implementation of graphalg. We defer better integration with query languages to future work. Queries can also be written directly in IPR, the internal representation of queries in AvantGraph, which does allow for full flexibility of the algorithm inputs. Written in IPR, the query shown in Listing 3.22 may instead be written as:

Listing 3.23: An IPR query with an embedded graph algorithm.

```
call(
    "
        func TriangleCount(graph: Matrix<bool>) -> int {
            L = select(tril, graph, -1);
            U = select(triu, graph, 1);
            C<L, struct> = L (+.one) U.T;

            return reduce(+, C);
        }
    ",
    (%val) = "TriangleCount" (
        matrix(
            projection(
                access(%0, "friend"), {
                    %row = src(%0),
                    %col = trg(%0),
                    %val = %0.weight,
                }
            ),
            %row, %col, %val),
    ),
)
```

This concludes our overview of the graphalg language design. In the following chapters, we discuss the implementation of the backends.

Chapter 4

Compiler Architecture

Our graphalg compiler has a parser and IR that both backends use. Each backend takes common graphalg IR as input and converts it to another internal IR. In Figure 4.1 we show a high-level overview of the compiler architecture.

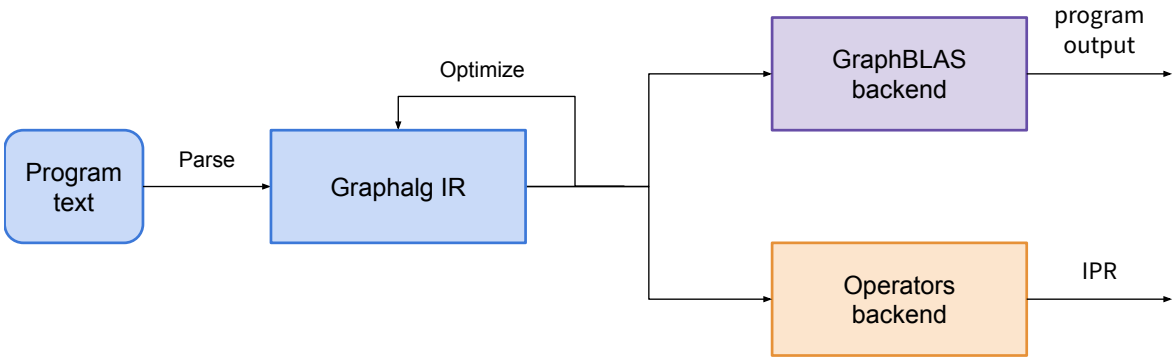


Figure 4.1: Graphalg compiler architecture.

AvantGraph uses ANTLR to parse queries. We have adopted the same strategy and developed an ANTLR grammar for graphalg. After ANTLR has constructed the parse tree we immediately convert it to graphalg IR. Graphalg IR is expressed in MLIR as a combination of the built-in dialects and a custom dialect that provides linear algebra operations. We cover graphalg IR in detail in Section 4.1.

Once the IR has been constructed we perform standard optimizations such as constant folding and dead-code elimination. These normalization rules are built-in to MLIR and are run by invoking the canonicalization pass. Canonicalization simplifies the IR if the program as the user wrote it contained code that can be simplified further, such as the statement `one_mil = 1000 * 1000;`. An additional advantage is that because canonicalization runs directly after parsing, the parser can focus on generating correct rather than efficient IR. One example of this is that when converting for loops, the parser conservatively assumes that all live variables are modified in the loop. When the canonicalization pass runs, it scans the loop body for loop-carried variables that are not modified inside the body and removes them.

After performing the optimizations useful for both backends, we hand off the IR to either the GraphBLAS or the operators backend. The GraphBLAS backend converts the IR into bytecode with calls to GraphBLAS for linear algebra operations. An interpreter executes the generated bytecode to produce the program output, which is then returned to the AvantGraph runtime to be used as an intermediate result in the query. For more details on the GraphBLAS backend, see Chapter 5. The operators backend does not evaluate the program, but instead transforms the graphalg IR into IPR so it can be fully integrated with the query and executed by the regular AvantGraph runtime. Its implementation is discussed in

Chapter 6. While we leave the implementation details of the backends for later chapters, this chapter does discuss how the backends are integrated into the AvantGraph query pipeline (Section 4.2).

4.1 Graphalg IR

In the design of graphalg IR we follow the following principles:

- Reuse built-in operations and types as much as possible. Not only does this avoid defining many operations ourselves, but it also allows us to reuse existing rewrite rules. For example, by using control flow operations included in MLIR, we get canonicalization rules for free.
- Break complex statements into simple operations. The graphalg syntax can encode complex operations in a single statement. Consider for example the statement `R<!M, struct> = A (min.+) B.T;`. In the GraphBLAS API, such a multiplication with a transposed input and mask with various flags set can be expressed with a single function call. For our IR we choose to break up such statements into multiple simpler operations to keep individual operations simple and avoid duplicate code. The GraphBLAS backend has additional optimization rules to fuse multiple simple operations into one efficient GraphBLAS function call.

We begin our overview of the IR with the supported types.

4.1.1 Data types

Almost all types in graphalg IR are taken from the built-in dialect:

- `bool`: `IntegerType`, width 1 (`i1`).
- `int`: `IntegerType`, width 64 (`i64`).
- `real`: `Float64Type` (`f64`).
- `index`: `IndexType` (`index`).
- `Vector<T>`: `RankedTensorType`, rank 1 (`tensor<?xT>`).
- `Matrix<T>`: `RankedTensorType`, rank 2 (`tensor<?x?xT>`).

MLIR does not have a built-in construct for ranges as they exist in graphalg (e.g. `0:42`), so we define a custom `RangeType` for operations that return a range.

4.1.2 Functions

We define a custom `FuncOp` and `ReturnOp` for graphalg IR. Both ops closely follow their counterparts from the `func` dialect in their design, but accept only the types defined above for parameters and return values. Additionally, `ReturnOp` is restricted to a single return value. Below in Listing 4.1 we give the IR for a simple function.

Listing 4.1: IR for a simple function.

```
// Original signature:
// func MyFunction(a: Matrix<int>, b:real) -> bool
graphalg.func @MyFunction(%arg0: tensor<?x?xi64>, %arg1: f64) -> i1 {
    %false = arith.constant false
    graphalg.return %false : i1
}
```

4.1.3 Control Flow

We do not define any new control flow operations for graphalg IR. Instead, we reuse the `ForOp` and `IfOp` from the `scf` dialect. If statements in graphalg map directly to an `IfOp` in the IR, as do for loops, provided that they do not have an `until` condition.

In the case of a loop with a break condition, we translate it to a `ForOp` with an inner `IfOp`. This means that after parsing, the two programs shown in Listing 4.2 result in equivalent IR, shown in Listing 4.3.

Listing 4.2: A program with a loop break condition.

```
func Until(n: int) -> int {
    for i in 0:100 until n > 10 {
        n = n + 1;
    }
    return n;
}

func UntilDesugared(n:int) -> int {
    for i in 0:100 {
        if n > 10 {
            // do nothing
        } else {
            n = n + 1;
        }
    }
    return n;
}
```

Listing 4.3: IR for a program with a loop break condition.

```
graphalg.func @Until(%arg0: i64) -> i64 {
      %c1_i64 = arith.constant 1 : i64
      %c10_i64 = arith.constant 10 : i64
      %idx1 = index.constant 1
      %c100_i64 = arith.constant 100 : i64
      %c0_i64 = arith.constant 0 : i64
      %0 = index.castu %c0_i64 : i64 to index
      %1 = index.castu %c100_i64 : i64 to index
      %2 = scf.for %arg1 = %0 to %1 step %idx1 iter_args(%arg2 = %arg0) -> (i64) {
            %3 = arith.cmpi sgt, %arg2, %c10_i64 : i64
            %4 = scf.if %3 -> (i64) {
                  scf.yield %arg2 : i64
            } else {
                  %5 = arith.addi %arg2, %c1_i64 : i64
                  scf.yield %5 : i64
            }
            scf.yield %4 : i64
      }
      graphalg.return %2 : i64
}
```

This representation does require the backends to recognize this pattern and avoid needlessly running a loop for many iterations, but the advantage is that all canonicalization rules from the `scf` are available. We have found this to be a worthwhile tradeoff.

4.1.4 Scalar Arithmetic

For scalar arithmetic, we use the `arith` and `index` dialects, without any custom extensions. Specifically, we use the following operations:

- `arith::ConstantOp` for constant values.
- `arith::SIToFPOp` for casting from `int` to `real`,
- `arith::FPToSIOp` for casting from `real` to `int`
- `index::CastUOp` for converting between `int` and `index` (both ways).
- `AddIOp`, `SubIOp`, `MulIOp`, `DivIOp` (and floating point equivalents) from `arith` for basic arithmetic.
- `arith::CmpIOp` and `arith::CmpFOp` for comparing scalars. Values of type `index` are cast to integers before comparing them.

4.1.5 Linear Algebra

Almost all operations used for linear algebra are custom operations added in the graphalg dialect (exceptions are `DimOp` and `EmptyOp`). We give an overview of them below.

SelectOp

`SelectOp` encodes the `select` expression. It applies a predicate to all entries of the input tensor and returns a new tensor only the entries for which the predicate holds. The IR in Listing 4.4 encodes the expression `select(>=, arg0, 0)`.

Listing 4.4: `select` example IR.

```
%c0_i64 = arith.constant 0 : i64
%0 = graphalg.select gte %arg0 : tensor<?x?xi64> %c0_i64 : i64 -> tensor<?x?xi64>
```

ApplyUnaryOp and ApplyBinaryOp

The `apply` expression can be used with binary unary and binary operators. In the IR we create separate operations for both variants, `ApplyUnaryOp` and `ApplyBinaryOp`. Below we give the IR for the expressions `apply(abs, arg0)` and `apply(+, arg0, 1)`.

Listing 4.5: `apply` example IR.

```
// apply(abs, arg0)
%0 = graphalg.apply_unary abs %arg0 : tensor<?x?xi64> -> tensor<?x?xi64>

// apply(+, arg0, 1)
%c1_i64 = arith.constant 1 : i64
%0 = graphalg.apply_binary add
    %arg0 : tensor<?x?xi64>
    %c1_i64 : i64 -> tensor<?x?xi64>
```

ElementWiseOp

Graphalg provides an element-wise operator with set intersection semantics and another with set union semantics. In the IR both variants are represented by an `ElementWiseOp`, with a flag to switch between intersection and union mode.

Listing 4.6: Element-wise example IR.

```
// arg0 .+ arg1
%0 = graphalg.ewise add
    %arg0 : tensor<?x?xi64>
    %arg1 : tensor<?x?xi64> set_intersection -> tensor<?x?xi64>

// add(+, arg0, arg1)
%0 = graphalg.ewise add
    %arg0 : tensor<?x?xi64>
    %arg1 : tensor<?x?xi64> set_union -> tensor<?x?xi64>
```

SemiringOp

Matrix multiplication with a semiring maps straightforwardly onto a dedicated `SemiringOp`. Because both matrix and vector types are represented by the tensor type in the IR, no separate operations are necessary for matrix-matrix, vector-matrix and matrix-vector specializations.

Listing 4.7: Matrix multiplication example IR.

```
// arg0 (min.+) arg1
%0 = graphalg.semiring
    (%arg0 : tensor<?x?xi64>)
    (min add)
    (%arg1 : tensor<?x?xi64>) -> tensor<?x?xi64>
```

TransposeOp

The `.T` property on matrices is represented in the IR with the `TransposeOp`.

Listing 4.8: Transpose example IR.

```
// arg0.T
%0 = graphalg.transpose %arg0 : tensor<?x?xi64> -> tensor<?x?xi64>
```

AccumOp

`AccumOp` copies all elements from a base tensor, and adds to that the elements of another tensor. If an entry is present in both matrices, the two values are merged using a binary operator. This operation is used to represent accumulation statements such as `arg0 += arg1`.

Listing 4.9: Accumulate example IR.

```
// arg0 += arg1
 %0 = graphalg.accum %arg0 : tensor<?x?xi64> add =
    %arg1 : tensor<?x?xi64> -> tensor<?x?xi64>
```

MaskOp

`MaskOp` encodes masked assignment statements such as `C<M> = A`. The operation has flags for making the mask structural, complemented and/or to ignore values in the base matrix (replace).

Listing 4.10: Masked assignment example IR.

```
// arg0<arg1> = arg2
%1 = graphalg.mask %arg0 : tensor<?x?xi64><%arg1 : tensor<?x?xi1>> =
    %0 : tensor<?x?xi64> -> tensor<?x?xi64> {
        complement = false,
        replace = false,
        structural = false
}

// arg0<!arg1, struct> = arg2
%1 = graphalg.mask %arg0 : tensor<?x?xi64><%arg1 : tensor<?x?xi1>> =
    %0 : tensor<?x?xi64> -> tensor<?x?xi64> {
        complement = true,
        replace = false,
        structural = true
}
```

ReduceOp and ReduceRowsOp

Graphalg IR has dedicated operations for reducing a matrix to a vector or a scalar.

Listing 4.11: `reduce` and `reduceRows` example IR.

```
// reduce(+, arg0)
%0 = graphalg.reduce add %arg0 : tensor<?x?xi64> -> i64

// reduceRows(+, arg0)
%0 = graphalg.reduce_rows add %arg0 : tensor<?x?xi64> -> tensor<?xi64>
```

Ranges

In graphalg IR, three kinds of ranges can be constructed:

- Singleton range: this represents a range consisting of a single element. It is used when assigning a scalar to a single element of a matrix.
- Bounded range: an index range with explicit start and endpoints. It is used to assign a value to multiple (but not all) elements of a matrix and as bounds of a for loop.
- 'All' range: an index range that represents an unbounded range. Used to fill all entries of a matrix with the same scalar value.

To create ranges, graphalg IR provides SingletonRangeOp, BoundedRangeOp and AllRangeOp.

Listing 4.12: Ranges example IR.

```
%c10_i64 = arith.constant 10 : i64
%c1_i64 = arith.constant 1 : i64
%c0_i64 = arith.constant 0 : i64
```

```
// [0]
%0 = index.castu %c0_i64 : i64 to index
%1 = graphalg.singleton_range %0

// [0:10]
%3 = index.castu %c0_i64 : i64 to index
%4 = index.castu %c10_i64 : i64 to index
%5 = graphalg.bounded_range %3 %4

// [:]
%7 = graphalg.all_range
```

AssignOp

`AssignOp` encodes the assignment of a matrix or scalar value to a submatrix of the given base matrix. This operation is free of side effects like all other operations in the dialect and therefore does not modify the base tensor. Instead, it returns a new tensor with the updated matrix.

`AssignOp` can also be used as a memory management hint to try and reuse an existing buffer for a new value. Such hints are used by the GraphBLAS backend to improve the bufferization strategy. They are ignored by the operators backend.

We show examples of both uses in Listing 4.13.

Listing 4.13: Assign example IR.

```
%c10_i64 = arith.constant 10 : i64
%c42_i64 = arith.constant 42 : i64
%c0_i64 = arith.constant 0 : i64

// arg0[0] = arg1;
%0 = index.castu %c0_i64 : i64 to index
%1 = graphalg.singleton_range %0
%2 = graphalg.assign %arg0 : tensor<?x?xi64>[%1] =
    %arg1 : tensor<?xi64> -> tensor<?x?xi64>

// arg0[0:10][:] = 42;
%0 = index.castu %c0_i64 : i64 to index
%1 = index.castu %c10_i64 : i64 to index
%2 = graphalg.bounded_range %0 %1
%3 = graphalg.all_range
%4 = graphalg.assign %arg0 : tensor<?x?xi64>[%2] [%3] =
    %c42_i64 : i64 -> tensor<?x?xi64>

// arg0 = arg1 (memory management hint)
%0 = graphalg.assign %arg0 : tensor<?x?xi64> =
    %arg1 : tensor<?x?xi64> -> tensor<?x?xi64>
```

NvalsOp

`NvalsOp` counts the number of present entries in a given matrix. It is used to represent the `.nvals` property.

Listing 4.14: Example IR for `NvalsOp`.

```
// arg0.nvals
%0 = graphalg.nvals %arg0 : tensor<?x?xi64>
```

SwapOp and ClearOp

Memory hint statements `swap` and `clear` map directly onto IR operations `SwapOp` and `ClearOp`, respectively.

Listing 4.15: Example IR for `SwapOp` and `ClearOp`.

```
// swap arg0 arg1;
%0:2 = graphalg.swap %arg0 %arg1 : tensor<?x?xi64>

// clear arg0;
%0 = graphalg.clear %arg0 : tensor<?x?xi64> -> tensor<?x?xi64>
```

DimOp

To represent the matrix properties `.nrows` and `.ncols` in the IR, we reuse the existing operation `DimOp` defined in the `tensor` dialect.

Listing 4.16: Example IR for `DimOp`.

```
// arg0.nrows
%c0 = arith.constant 0 : index
%dim = tensor.dim %arg0, %c0 : tensor<?x?xi64>

// arg0.ncols
%c1 = arith.constant 1 : index
%dim = tensor.dim %arg0, %c1 : tensor<?x?xi64>
```

EmptyOp

Another operation we can reuse from the `tensor` dialect is `EmptyOp`, which creates an empty tensor. `EmptyOp` is used to encode the creation of a new matrix or vector.

Listing 4.17: Example IR for `EmptyOp`.

```
// M = matrix<int>(42, 42);
%c42_i64 = arith.constant 42 : i64
%0 = index.castu %c42_i64 : i64 to index
%2 = tensor.empty(%0, %0) : tensor<?x?xi64>
```

This concludes our overview of operations used in the IR. In the next section, we discuss how the graphalg compiler is integrated into the query pipeline.

4.2 Integration with AvantGraph

Although both backends take the same IR as input, their output is very different, which also reflects in their placement in the query pipeline. In Figure 4.2 we show how the backends are integrated into the AvantGraph query pipeline.

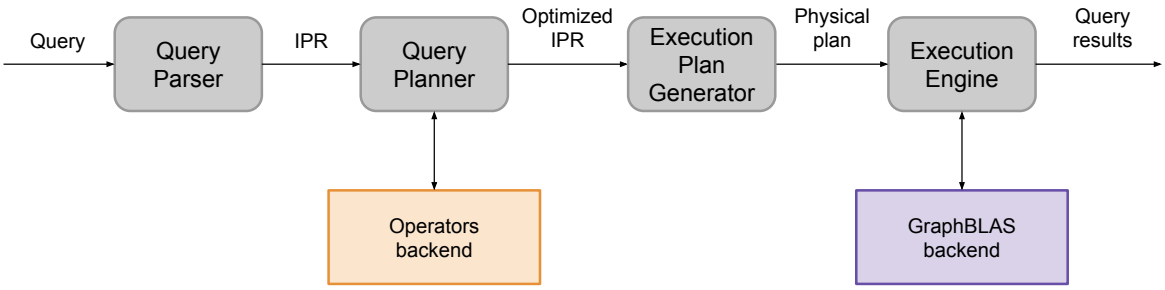


Figure 4.2: Integration of backends into the query pipeline.

4.2.1 GraphBLAS backend

The GraphBLAS backend integrates AvantGraph at a late stage of the pipeline, during query execution. Throughout the planning and execution plan generation, the algorithm is stored as a string and is not analyzed by AvantGraph at all. In the final execution plan, the graph algorithm invocation is represented by a *Call operator* we add to AvantGraph for this specific purpose. The call operator first gathers all blocks from its input relations and loads them in the GraphBLAS data format. Only once all inputs to the algorithm have been fully evaluated does it start to process the algorithm. The interpreter evaluates the program and returns the output, also in the GraphBLAS data format. The call operator converts the output into blocks of tuples and sends them to its output relation.

Coupling with the database is loose in the case of GraphBLAS, and there is a clear separation of concerns between the execution of the algorithm and the query. The downsides are a considerable overhead from converting between data formats, and little opportunity for cross-optimization.

4.2.2 Operators backend

The approach taken by the operators backend is radically different. The backend integrates with AvantGraph very early on in the pipeline, right after the query has been parsed into IPR. Before the query planner performs any normalization or optimization steps, the algorithm is passed to the operators backend. There, the algorithm is parsed to graphalg IR and transformed into IPR. The generated IPR replaces the algorithm invocation in the query, yielding a query plan with an embedded algorithm that is fully expressed in IPR. To the remainder of the pipeline, there is no distinction between the query and the algorithm.

Thanks to the unified representation, the algorithm can be optimized together with the query, and there is no runtime communication overhead between the query and the algorithm. The price we pay for this tight coupling is additional engineering work to map linear algebra operations to relational algebra, and in some cases implement operations from scratch. We are also forced to use the internal data format of unordered streams of tuples, instead of a more efficient format designed specifically for matrices.

4.2.3 Important differences

In the sections above we have already alluded to the main differences between the two backends. The most important is the level of *visibility* the database has into the algorithm. In the operators backend the algorithm is converted into IPR, giving AvantGraph full visibility into the details of the algorithm. This allows the pipeline to perform simplification of the algorithm, plan data access, and optimize it together with the rest of the query. The contrast with the GraphBLAS backend could not be more different: the database has no visibility

at all into the algorithm, it just passes a string to the backend and receives an output. The upside is that the GraphBLAS backend is easier to port to other database systems.

Another important difference is that the operators backend adopts a *stream processing* approach, whereas the GraphBLAS backends is closer to *batch processing*. Because intermediate results between operators can be large, AvantGraph tries to pipeline query execution where possible to reduce memory consumption. The operators backend converts to IPR, so algorithms will naturally take advantage of AvantGraph's pipelining. In the GraphBLAS backend, however, a data format conversion is required before the algorithm can be executed, so inputs must be fully buffered before executing the algorithm. This leads to higher loading times and memory requirements, but once the data has been converted into the GraphBLAS format, we can use readily available and highly-optimized linear algebra routines.

This concludes the high-level overview of the graphalg compiler and backends. We discuss the GraphBLAS backend in detail in the next chapter.

Chapter 5

Compilation to GraphBLAS

This chapter covers the GraphBLAS backend for graphalg. It is a black-box batch style integration with AvantGraph: all input data is converted from AvantGraph's native storage format into GraphBLAS data types. AvantGraph has no visibility into the algorithm and does not perform any optimizations on it, relying fully on the GraphBLAS library.

The GraphBLAS backend transforms graphalg programs into a simple IR with a direct mapping to GraphBLAS library calls. Arguments to the program are provided either as plain scalar values, or as matrices in the native GraphBLAS format. An interpreter takes these arguments and executes the final IR to produce the output. An overview of the compilation pipeline is provided in Figure 5.1.

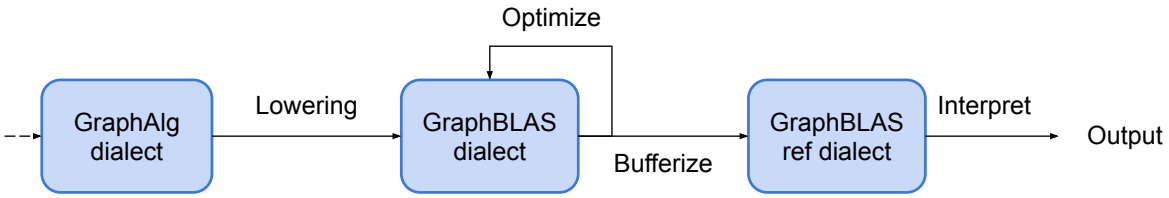


Figure 5.1: Pipeline of the GraphBLAS backend.

In the next sections, we discuss the pipeline in more detail:

- Conversion of the graphalg dialect into the GraphBLAS dialect (lowering) is covered in Section 5.1.
- After this conversion step an optimization pass attempts to fuse linear algebra operations as much as possible (Section 5.2).
- Another conversion is then applied, this time to convert from value semantics into reference semantics for matrices. This step is called *bufferization*, and is described in Section 5.3.
- Finally, in Section 5.5 we discuss the last step of the pipeline, program interpretation.

5.1 Lowering to GraphBLAS

This lowering pass converts all operations from the graphalg dialect into a new GraphBLAS dialect. The GraphBLAS dialect is design to resemble the GraphBLAS API (with SuiteSparse [12] extensions where necessary) as much as possible for easy interpretation later (see Section 5.5). All other operations unrelated to linear algebra such as scalar arithmetic and control flow are unaffected by the lowering: the interpreter can execute them directly. Below

we describe how each of the linear algebra operations are lowered into GraphBLAS operations. Because the design of graphalg was heavily inspired by GraphBLAS, many lowerings are a one-to-one mapping of operations.

5.1.1 SemiringOp

SemiringOps correspond to the `GrB_mxm`, `GrB_vxm` or `GrB_mxv` function in the GraphBLAS API, depending on the rank of the arguments. Because the rank of the inputs is already encoded into the argument types and the methods are otherwise equivalent, we choose to define a single SemiringOp in the GraphBLAS dialect and lower to that. The final decision on which of the GraphBLAS methods to invoke is made by the interpreter based on the argument type information. The GraphBLAS SemiringOp is similar to the graphalg one, but adds additional parameters that are passed in the GraphBLAS library call:

- `base`: base matrix used with mask or accumulator. If a mask is given, entries of `base` outside the mask are copied to the output. If an accumulator is set, all entries of `base` are copied to the output and merged with entries from the multiplication result.
- `mask`: defines the output positions for which a matrix multiplication should be computed. It is equivalent to computing the full result and then applying a graphalg MaskOp.
- `accum`: the accumulator monoid to use when merging entries from `base` with the multiplication result.
- `descriptor`: contains flags to transpose one of the inputs, complement the mask, or make the mask structural. There is also a `replace` flag available which, when combined with `base`, prevents any copying of elements from `base`. Specifying `replace` and `base` is therefore functionally equivalent to specifying neither. Its use is to suggest to the bufferization pass (Section 5.3) that it should try to use the buffer of `base` to store the result.

A graphalg SemiringOp is lowered into a GraphBLAS SemiringOp by copying all operands and leaving the additional parameters unset. The new parameters are still needed in the IR because the later optimization pass, described in Section 5.2, may rewrite the operation and set a value for these new parameters.

5.1.2 ElementWiseOp

The lowering of ElementWiseOps follows a very similar pattern to the SemiringOp. They map to either `GrB_eWiseMult` or `GrB_eWiseAdd`, depending on whether the join mode is set to intersection or union, respectively. The same parameters `base`, `mask`, `accum` and `descriptor` are introduced, and are left unset by the lowering.

5.1.3 ApplyUnaryOp and ApplyBinaryOp

Both operations are mapped directly to an equivalently named op in the GraphBLAS dialect. The new operations have the same parameters as their graphalg counterparts, once again extended with `base`, `mask`, `accum` and `descriptor`. These operations map to the GraphBLAS function `GrB_apply`.

5.1.4 SelectOp

SelectOp is directly mapped to an equivalent operation with the same four additional parameters left unset. It corresponds to `GrB_select`. `GrB_select` is not one but a family of functions with different matrices types (`GrB_Matrix` and `GrB_Vector`) and different types for the scalar input. The interpreter calls the correct function based on MLIR's explicit argument types.

5.1.5 ReduceOp

ReduceOp is mapped to an operation of the same name in the GraphBLAS dialect. It is not extended with any additional parameters because its return value is a scalar. The conversion is thus purely syntactical, both versions have identical operands and attributes. The operation maps to `GrB_reduce`.

5.1.6 ReduceRowsOp

Conversion of ReduceRowsOp is similar to ReduceOp, but because the result is a `Vector<T>`, the usual parameters `base`, `mask`, `accum` and `descriptor` are added. The corresponding function is `GrB_Matrix_reduce_Monoid`.

5.1.7 AssignOp

AssignOp is straightforwardly mapped to a GraphBLAS AssignOp. Since the graphalg AssignOp already has `base`, it is extended only with `mask`, `accum` and `descriptor`. The GraphBLAS library function `GrB_assign` supports all ranges available in graphalg, so these are similarly trivial to convert. AssignOp is a relatively simple operation, particularly if all ranges are set to an AllRangeOp, in which case it represents a simple copy. Because the GraphBLAS version has an optional mask and accumulator, it is also a suitable target for the next two operations.

5.1.8 AccumOp

The GraphBLAS dialect has no direct analog to AccumOp. However, because AssignOp has an accumulator setting, it can represent an AccumOp by setting `base` and `accum`. Since most linear algebra operations have an `accum` field, the AssignOp can typically be fused with the operation that produces the value to accumulate. See Section 5.2 for more details.

Below we show an example of a program with an AccumOp that cannot be fused to another operation.

Listing 5.1: A function containing an isolated accumulate operation

```
func Accum(a: Vector<int>, b: Vector<int>) -> Vector<int> {
    a += b;
    return a;
}
```

The AccumOp is therefore lowered into an AssignOp. The corresponding IR is given in Listing 5.2.

Listing 5.2: Lowered IR for Listing 5.1

```
func.func @Accum(%arg0: tensor<?xi64>, %arg1: tensor<?xi64>) -> tensor<?xi64> {
    %0 = mlir_graphblas.all_range
    %1 = mlir_graphblas.assign %arg0 : tensor<?xi64> [%0] add =
            %arg1 : tensor<?xi64> -> tensor<?xi64> {
                complement = false,
```

```
                replace = false,
                structural = false,
                transpose = false
            }
    return %1 : tensor<?xi64>
}
```

5.1.9 MaskOp

The same strategy is adopted for MaskOp. MaskOps are lowered to AssignOps by setting the `mask` flag.

5.1.10 TransposeOp

The GraphBLAS dialect has a TransposeOp with `base`, `mask`, `accum` and `descriptor`, to match the signature of `GrB_transpose`. In practice, we have not found any programs where the additional parameters are useful. Therefore, our backend emits only plain transpose operations with all optional parameters unset, and relevant transformations assert that these fields remain unset.

5.1.11 SwapOp and ClearOp

We replace the SwapOp and ClearOp operations with identical operations defined in the GraphBLAS dialect. The operations do not undergo any real transformation: we only replace them to ensure that the IR after this pipeline stage is free of references to the graphalg dialect. After this, the operations are preserved until they are used in the Bufferization stage (see Section 5.3).

5.1.12 NvalsOp

An NvalsOp maps to `GrB_Matrix_nvals` or `GrB_Vector_nvals`, depending on the type of `value`. Like SwapOp and ClearOp it is replaced by an identical operation in the GraphBLAS dialect. Unlike these operations, however, NvalsOp is preserved until the final IR, and directly executed by the interpreter.

5.1.13 DimOp

DimOps from the `tensor` dialect are mapped to equivalent operations from the `memref` dialect during bufferization. Memref DimOps are executed directly by the interpreter as one of one of `GrB_Matrix_nrows`, `GrB_Matrix_ncols` or `GrB_Vector_nrows`, depending on the type of `source` and the dimension requested in `index`.

5.1.14 EmptyOp

EmptyOps that remain after optimization are converted to AllocOps (from the `memref` dialect). Depending on the requested dimensions, the interpreter executes an AllocOp by calling either `GrB_Matrix_new` or `GrB_Vector_new`.

5.2 Optimization

The lowering transformation described above produces an IR that can be immediately bufferized, but the resulting program will likely take a long time to execute and consume excessive

memory in the process. To illustrate the problem, consider the graphalg program given in Listing 5.3.

Listing 5.3: A tiny graphalg program.

```
func Example(M:Matrix<bool>, A:Matrix<int>, B:Matrix<int>) -> Matrix<int> {
    C<M> = A (+.*) B;
    return C;
}
```

After parsing to the graphalg dialect, we obtain the IR shown in Listing 5.4.

Listing 5.4: Program IR directly after parsing

```
graphalg.func @Example(
            %arg0: tensor<?x?xi1>,
            %arg1: tensor<?x?xi64>,
            %arg2: tensor<?x?xi64>) -> tensor<?x?xi64> {
      %0 = graphalg.semiring
            (%arg1 : tensor<?x?xi64>)
            (add mul)
            (%arg2 : tensor<?x?xi64>) -> tensor<?x?xi64>
      %1 = graphalg.mask<%arg0 : tensor<?x?xi1>> =
            %0 : tensor<?x?xi64> -> tensor<?x?xi64>
            {complement = false, replace = false, structural = false}
      graphalg.return %1 : tensor<?x?xi64>
}
```

The steps of computing the matrix product and applying the mask are modeled as separate operations. In Listing 5.5 we see that lowering maps the instructions one-to-one.

Listing 5.5: Program IR after lowering to the GraphBLAS dialect

```
func.func @Example(
            %arg0: tensor<?x?xi1>,
            %arg1: tensor<?x?xi64>,
            %arg2: tensor<?x?xi64>) -> tensor<?x?xi64> {
      %0 = mlir_graphblas.semiring null =
            (%arg1 : tensor<?x?xi64>)
            (add mul)
            (%arg2 : tensor<?x?xi64>) -> tensor<?x?xi64> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transposeLeft = false,
                  transposeRight = false
            }
      %1 = mlir_graphblas.all_range
      %2 = mlir_graphblas.all_range
      %3 = mlir_graphblas.assign <%arg0 : tensor<?x?xi1>> [%1, %2] null =
            %0 : tensor<?x?xi64> -> tensor<?x?xi64> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transpose = false
            }
```

```
      return %3 : tensor<?x?xi64>
}
```

Directly executing this IR may be inefficient because it requires that we first compute the full matrix product of `A` and `B`, then filter out all results not in `M`. If `M` is very sparse, a much faster approach is to first check which entries are `true` in `M`, and only compute those entries of `C`. It is for this reason that `GrB_mxm` has a `mask` parameter: it allows fusing the masked assignment with the computation of a matrix product.

5.2.1 Assign Fusion

To exploit this feature of the GraphBLAS API, we develop an optimization rewrite rule to fuse AssignOps with any op that has `base`, `mask` and `descriptor`. As a result, our compiler optimizes Listing 5.5 into the IR given in Listing 5.6.

Listing 5.6: Program IR directly after AssignOp fusion

```
func.func @Example(
            %arg0: tensor<?x?xi1>,
            %arg1: tensor<?x?xi64>,
            %arg2: tensor<?x?xi64>) -> tensor<?x?xi64> {
      %0 = mlir_graphblas.semiring <%arg0 : tensor<?x?xi1>> null =
            (%arg1 : tensor<?x?xi64>)
            (add mul)
            (%arg2 : tensor<?x?xi64>) -> tensor<?x?xi64> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transposeLeft = false,
                  transposeRight = false
            }
      return %0 : tensor<?x?xi64>
}
```

The semiring operation subsumes the assign operation by incorporating the mask matrix. The body of the example function contains a single operation that maps directly to `GrB_mxm`, which is efficient as can be within the constraints of the GraphBLAS API. In the given example there is no base matrix to assign to, but our optimization rules can also fuse the ops if `C` is an existing matrix.

5.2.2 Transpose Fusion

Transpose operations are another prime candidate for fusion. Consider the program shown in Listing 5.7.

Listing 5.7: A matrix product where the right-hand side is transposed

```
func ExampleT(A:Matrix<int>, B:Matrix<int>) -> Matrix<int> {
    C = A (+.*) B.T;
    return C;
}
```

Depending on the representation used for the matrix `B`, transposing it can be a very expensive operation to perform. If the transpose of `B` is only needed to perform the matrix multiplication, we may be able to avoid computing it entirely. Recall that the definition of matrix multiplication is:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

To adjust the formula for the case where the right-hand side is transposed, we swap indices k and j:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{jk}$$

Based on this formula we can conclude that so long as the representation of `B` (no transposed) provides efficient row-wise access, transposing it is not necessary. Moreover, if `B` provides fast row-wise access but reading columns is slow, computing `A (+.*) B.T` may even be faster than computing `A (+.*) B`. Once again, the GraphBLAS API accounts for this with extra parameters on `GrB_mxm`. In `descriptor` we can set `transposeRight` to indicate that the right input to the multiplication should be transposed first. A rewrite rule checks if the left or right input to a SemiringOp is a TransposeOp, and removes the TransposeOp by flipping the flag in SemiringOp. Listing 5.8 shows the change in IR resulting from this optimization (the source program was shown in Listing 5.7).

Listing 5.8: Example of transpose fusion.

```
// Original lowered IR
func.func @ExampleT(
            %arg0: tensor<?x?xi64>,
            %arg1: tensor<?x?xi64>) -> tensor<?x?xi64> {
      %0 = mlir_graphblas.transpose null =
            transpose(%arg1 : tensor<?x?xi64>) -> tensor<?x?xi64> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transpose = false
            }
      %1 = mlir_graphblas.semiring null =
            (%arg0 : tensor<?x?xi64>)
            (add mul)
            (%0 : tensor<?x?xi64>) -> tensor<?x?xi64> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transposeLeft = false,
                  transposeRight = false
            }
      return %1 : tensor<?x?xi64>
}

// Optimized IR
func.func @ExampleT(
            %arg0: tensor<?x?xi64>,
            %arg1: tensor<?x?xi64>) -> tensor<?x?xi64> {
      %0 = mlir_graphblas.semiring null =
            (%arg0 : tensor<?x?xi64>)
            (add mul)
            (%arg1 : tensor<?x?xi64>) -> tensor<?x?xi64> {
```

```
                complement = false,
                replace = false,
                structural = false,
                transposeLeft = false,
                transposeRight = true
            }
        return %0 : tensor<?x?xi64>
}
```

5.2.3 Further Optimization Opportunities

In our experiments, we have found the two optimizations described to be sufficient for turning idiomatic graphalg programs into efficient GraphBLAS IR. The generated code is comparable in performance to hand-optimized C versions. Still, we believe there are plenty of opportunities for further optimization, which we leave as future work. For example, the current backend performs little optimization to avoid redundant work. Consider the program below, which first reduces a matrix to a vector, then reduces the vector down to a scalar.

Listing 5.9: Reducing to a vector, then reducing to a scalar.

```
func ReduceReduce(M:Matrix<int>) -> int {
      V = reduceRows(+, M);
      return reduce(+, V);
}
```

This program can be improved by omitting the `reduceRows` call and directly reducing the entries of the matrix instead. For a less experienced programmer in particular, the inefficiency of this program may not be obvious. Instead of requiring the programmer to make this optimization, it could be implemented in the compiler as a canonicalization rule in the graphalg dialect, where it would also benefit the other backend. The programs we use for our evaluation do not have such inefficiencies, and therefore we have not implemented this optimization.

5.3 Bufferization

In the bufferization pass, operations with value semantics are converted into equivalent operations with reference semantics. Allocations and deallocations for buffers are also made explicit. Bufferization is an optimization problem similar to register allocation. The amount of buffers allocated is to be minimized, as is the copying of data between buffers. Existing buffers should be used as much as possible, but care must be taken to avoid overwriting contents that are still required for later computation.

To implement bufferization of our IR, we use the standard MLIR bufferization infrastructure [52]. While it has been designed for use with flat memory buffers, we have found that it works equally well for GraphBLAS matrices. The bufferization framework uses the `memref` type for buffers, which our interpreter treats as a pointer to a GraphBLAS object. Through the `BufferizableOpInterface`, support for custom operations can be added to the framework. We have implemented this interface for all GraphBLAS operations. The built-in dialects of MLIR already implement this interface. Even complex operations like loops can be bufferized without any additional effort on our part.

Below we give an example of a simple function. The corresponding IR before and after bufferization is given in Listing 5.11.

Listing 5.10: A simple function to bufferize

```
func BufferizeMe() -> int {
    v = Vector<int>(10);
    v[:] = 42;
    return reduce(+, v);
}
```

Listing 5.11: IR before bufferization

```
// Before bufferization
func.func @BufferizeMe() -> i64 {
      %c42_i64 = arith.constant 42 : i64
      %c10_i64 = arith.constant 10 : i64
      %0 = index.castu %c10_i64 : i64 to index
      %1 = tensor.empty(%0) : tensor<?xi64>
      %2 = mlir_graphblas.all_range
      %3 = mlir_graphblas.assign %1 : tensor<?xi64> [%2] null =
            %c42_i64 : i64 -> tensor<?xi64> {
            complement = false,
            replace = false,
            structural = false,
            transpose = false
      }
      %4 = mlir_graphblas.reduce null = reduce(add, %3 : tensor<?xi64>) -> i64
      return %4 : i64
}

// After bufferization
func.func @BufferizeMe() -> i64 {
      %c42_i64 = arith.constant 42 : i64
      %c10_i64 = arith.constant 10 : i64
      %0 = index.castu %c10_i64 : i64 to index
      %alloc = memref.alloc(%0) {alignment = 64 : i64} : memref<?xi64>
      %1 = mlir_graphblas.all_range
      mlir_graphblas.assign_ref(
            %alloc : memref<?xi64>,
            GrB_NULL,
            null,
            %1,
            %c42_i64 : i64) {
            complement = false,
            replace = false,
            structural = false,
            transpose = false
      }
      %2 = mlir_graphblas.reduce_ref(
            GrB_NULL,
            null,
            add,
            %alloc : memref<?xi64>) : i64
      memref.dealloc %alloc : memref<?xi64>
      return %2 : i64
}
```

During bufferization, the empty tensor op has been converted into a memory allocation. A deallocation op has also been added to free the memory once it is no longer needed. Both GraphBLAS operations have been replaced with equivalent ops that write their output to a `memref` instead of returning a new tensor.

While the MLIR bufferization pass has generally worked well for us, there are some important limitations. First of all, the algorithm is greedy, so it is not guaranteed to be optimal. Furthermore, the bufferization pass does not recycle buffers. If a buffer of size n is no longer needed, and the next instruction allocates a new buffer of size n, the algorithm does not currently reuse the existing buffer. For an example of this problem, consider the function below.

Listing 5.12: A simple loop that updates a variable

```
func Hints() -> Vector<int> {
    a = Vector<int>(10);
    for i in 0:10 {
      // update a
      a = ...;
    }
    return a;
}
```

The program is intended to allocate a vector `a` once and update it repeatedly inside the loop. Once the program has been converted in SSA form though, this intent is no longer obvious, as shown in Listing 5.13.

Listing 5.13: IR for Listing 5.12.

```
graphalg.func @Hints() -> tensor<?xi64> {
    %idx1 = index.constant 1
    %c0_i64 = arith.constant 0 : i64
    %c10_i64 = arith.constant 10 : i64
    %0 = index.castu %c10_i64 : i64 to index
    %1 = tensor.empty(%0) : tensor<?xi64>
    %2 = index.castu %c0_i64 : i64 to index
    %3 = index.castu %c10_i64 : i64 to index
    %4 = scf.for %arg0 = %2 to %3 step %idx1 iter_args(%arg1 = %1)
            -> (tensor<?xi64>) {
            %5 = /* update a */
            scf.yield %5 : tensor<?xi64>
    }
    graphalg.return %4 : tensor<?xi64>
}
```

A fresh variable `%5` is created for the updated value of a, with no obvious connection to `%arg1`. Without buffer recycling, every updated version of a creates a new allocation, and the old buffer is freed. To circumvent this issue, graphalg inserts additional AssignOps as a hint to the bufferization pass to try and reuse a particular buffer. In Listing 5.14, we show the IR as graphalg generates it, with an additional AssignOp.

Listing 5.14: IR for Listing 5.12, with AssignOp hint.

```
graphalg.func @Hints() -> tensor<?xi64> {
    %idx1 = index.constant 1
    %c0_i64 = arith.constant 0 : i64
    %c10_i64 = arith.constant 10 : i64
```

```
    %0 = index.castu %c10_i64 : i64 to index
    %1 = tensor.empty(%0) : tensor<?xi64>
    %2 = index.castu %c0_i64 : i64 to index
    %3 = index.castu %c10_i64 : i64 to index
    %4 = scf.for %arg0 = %2 to %3 step %idx1 iter_args(%arg1 = %1)
            -> (tensor<?xi64>) {
            %5 = /* update a */
            %6 = graphalg.assign %arg1 : tensor<?xi64> =
                  %5 : tensor<?xi64> -> tensor<?xi64>
            scf.yield %6 : tensor<?xi64>
    }
    graphalg.return %4 : tensor<?xi64>
}
```

To also enable buffer recycling with operations that only write to some entries of a matrix (e.g. masked assignment), we have introduced the `clear` operation to graphalg. The last memory hint operation, `swap`, is not related to recycling, but rather to a mismatch between GraphBLAS objects and the flat buffers that MLIR expects to bufferize for. With flat buffers, swapping two instances requires copying all elements of the buffers. For GraphBLAS objects in our interpreter, we can instead perform a cheap pointer swap. This is not something that can be encoded in the bufferization framework, so instead we add a custom operation for it.

For readers interested in more details about MLIR's bufferization architecture, we recommend watching [38]. The pass documentation [52] also gives a good overview of the design goals.

5.4 Example Pipeline

Now that we have discussed all stages of the compiler pipeline up to the interpreter, we show how these passes together transform graphalg programs into optimized GraphBLAS IR. As our running example, we take the simplest program from the GAP benchmark suite, triangle count:

Listing 5.15: Triangle count program in graphalg (from the GAP benchmark suite).

```
func TriangleCount(graph: Matrix<bool>) -> int {
    L = select(tril, graph, -1);
    U = select(triu, graph, 1);
    C<L, struct> = L (+.one) U.T;

    return reduce(+, C);
}
```

The first step is to parse the source text into graphalg IR. For the triangle count program above, the following IR is produced:

Listing 5.16: Parsed triangle count program.

```
graphalg.func @TriangleCount(%arg0: tensor<?x?xi1>) -> i64 {
      %c-1_i64 = arith.constant -1 : i64
      %c1_i64 = arith.constant 1 : i64
      %0 = graphalg.select tril
            %arg0 : tensor<?x?xi1> %c-1_i64 : i64 -> tensor<?x?xi1>
      %1 = graphalg.select triu
            %arg0 : tensor<?x?xi1> %c1_i64 : i64 -> tensor<?x?xi1>
      %2 = graphalg.transpose %1 : tensor<?x?xi1> -> tensor<?x?xi1>
```

```
      %3 = graphalg.semiring
            (%0 : tensor<?x?xi1>)
            (add one)
            (%2 : tensor<?x?xi1>) -> tensor<?x?xi64>
      %4 = graphalg.mask<%0 : tensor<?x?xi1>> =
            %3 : tensor<?x?xi64> -> tensor<?x?xi64> {
                  complement = false,
                  replace = false,
                  structural = true
            }
      %5 = graphalg.reduce add %4 : tensor<?x?xi64> -> i64
      graphalg.return %5 : i64
}
```

There is an almost one-to-one correspondence between statements in the source program and operations in the IR. The only exceptions are that the transpose of `U`, `U.T`, becomes its own operation, and the masked assignment is split from the semiring expression. In the next stage of the pipeline, the IR is lowered into the GraphBLAS dialect:

Listing 5.17: Lowered triangle count IR.

```
func.func @TriangleCount(%arg0: tensor<?x?xi1>) -> i64 {
      %c-1_i64 = arith.constant -1 : i64
      %c1_i64 = arith.constant 1 : i64
      %0 = mlir_graphblas.select null = select(
            tril,
            %arg0 : tensor<?x?xi1>,
            %c-1_i64 : i64) -> tensor<?x?xi1> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transpose = false
            }
      %1 = mlir_graphblas.select null = select(
            triu,
            %arg0 : tensor<?x?xi1>,
            %c1_i64 : i64) -> tensor<?x?xi1> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transpose = false
            }
      %2 = mlir_graphblas.transpose null = transpose(
            %1 : tensor<?x?xi1>) -> tensor<?x?xi1> {
                  complement = false,
                  replace = false,
                  structural = false,
                  transpose = false
            }
      %3 = mlir_graphblas.semiring null =
            (%0 : tensor<?x?xi1>)
            (add one)
            (%2 : tensor<?x?xi1>) -> tensor<?x?xi64> {
                  complement = false,
```

```
                replace = false,
                structural = false,
                transposeLeft = false,
                transposeRight = false
        }
    %4 = mlir_graphblas.all_range
    %5 = mlir_graphblas.all_range
    %6 = mlir_graphblas.assign<%0 : tensor<?x?xi1>> [%4, %5] null =
        %3 : tensor<?x?xi64> -> tensor<?x?xi64> {
                complement = false,
                replace = false,
                structural = true,
                transpose = false
        }
    %7 = mlir_graphblas.reduce null =
        reduce(add, %6 : tensor<?x?xi64>) -> i64
    return %7 : i64
}
```

Most operations have been mapped directly to their GraphBLAS counterparts. Because the GraphBLAS dialect has no mask operation, that operation has been mapped to `assign`. The IR is now ready for optimization. In this case, the transpose operation can be fused to the semiring, and the semiring can be fused with the assign. Performing the fusion yields the following optimized IR:

Listing 5.18: Optimized triangle count IR.

```
func.func @TriangleCount(%arg0: tensor<?x?xi1>) -> i64 {
    %c-1_i64 = arith.constant -1 : i64
    %c1_i64 = arith.constant 1 : i64
    %0 = mlir_graphblas.select null = select(
        tril,
        %arg0 : tensor<?x?xi1>,
        %c-1_i64 : i64) -> tensor<?x?xi1> {
            complement = false,
            replace = false,
            structural = false,
            transpose = false
        }
    %1 = mlir_graphblas.select null = select(
        triu,
        %arg0 : tensor<?x?xi1>,
        %c1_i64 : i64) -> tensor<?x?xi1> {
            complement = false,
            replace = false,
            structural = false,
            transpose = false
        }
    %2 = mlir_graphblas.semiring<%0 : tensor<?x?xi1>> null =
        (%0 : tensor<?x?xi1>)
        (add one)
        (%1 : tensor<?x?xi1>) -> tensor<?x?xi64> {
            complement = false,
            replace = false,
```

```
                structural = true,
                transposeLeft = false,
                transposeRight = true
        }
    %3 = mlir_graphblas.reduce null =
        reduce(add, %2 : tensor<?x?xi64>) -> i64
    return %3 : i64
}
```

As a final step before we can execute the program in the interpreter, the IR must be bufferized. The resulting IR is still similar to the optimized IR shown above but with memory allocation operations added.

Listing 5.19: Bufferized triangle count IR

```
func.func @TriangleCount(%arg0: memref<?x?xi1>) -> i64 {
    %c1 = arith.constant 1 : index
    %c-1_i64 = arith.constant -1 : i64
    %c1_i64 = arith.constant 1 : i64
    %c0 = arith.constant 0 : index
    %dim = memref.dim %arg0, %c0 : memref<?x?xi1>
    %dim_0 = memref.dim %arg0, %c1 : memref<?x?xi1>
    %alloc = memref.alloc(%dim, %dim_0) {alignment = 64 : i64} : memref<?x?xi1>
    mlir_graphblas.select_ref(
        %alloc : memref<?x?xi1>,
        GrB_NULL,
        null,
        tril,
        %arg0 : memref<?x?xi1>,
        %c-1_i64 : i64) {
            complement = false,
            replace = false,
            structural = false,
            transpose = false
        }
    %dim_1 = memref.dim %arg0, %c0 : memref<?x?xi1>
    %dim_2 = memref.dim %arg0, %c1 : memref<?x?xi1>
    %alloc_3 = memref.alloc(%dim_1, %dim_2)
        {alignment = 64 : i64} : memref<?x?xi1>
    mlir_graphblas.select_ref(
        %alloc_3 : memref<?x?xi1>,
        GrB_NULL,
        null,
        triu,
        %arg0 : memref<?x?xi1>,
        %c1_i64 : i64) {
            complement = false,
            replace = false,
            structural = false,
            transpose = false
        }
    %alloc_4 = memref.alloc(%dim, %dim_2)
        {alignment = 64 : i64} : memref<?x?xi64>
    mlir_graphblas.semiring_ref(
```

```
            %alloc_4 : memref<?x?xi64>,
            %alloc : memref<?x?xi1>,
            null,
            add,
            one,
            %alloc : memref<?x?xi1>,
            %alloc_3 : memref<?x?xi1>) {
                  complement = false,
                  replace = false,
                  structural = true,
                  transposeLeft = false,
                  transposeRight = true
            }
      %0 = mlir_graphblas.reduce_ref(
            GrB_NULL,
            null,
            add,
            %alloc_4 : memref<?x?xi64>) : i64
      memref.dealloc %alloc : memref<?x?xi1>
      memref.dealloc %alloc_3 : memref<?x?xi1>
      memref.dealloc %alloc_4 : memref<?x?xi64>
      return %0 : i64
}
```

This represents the final IR for the triangle count program. Memory allocation has been made explicit, and all linear algebra operations correspond directly to one the functions from the GraphBLAS API. It is ready to be executed by the interpreter, which we discuss next.

5.5 Interpreter

The interpreter is the final stage of the GraphBLAS backend pipeline. It takes in the bufferized IR along with the name of the function to execute and values for the parameters and executes the IR. The interpreter executes directly on the given IR and does not change it in any way. The implementation mostly follows the classical style using a big switch inside of a loop, as illustrated in Listing 5.20.

Listing 5.20: Illustration of the core interpreter loop.

```
// function is a reference to the function to execute
// results is a mapping from operations to their computed results.
for (auto op : function.getBody()) {
      switch (op.getType()) {
      case ADD_INT:
            // op.lhs and op.rhs are references
            // to the inputs of the add operation.
            results[op] = results[op.lhs] + results[op.rhs];
            break;
      }
      // ... more operations
      case RETURN:
            // op.value is a reference to the operation
            // result to return.
            return results[op.value];
```

```
}
```

5.5.1 Handling Control Flow

In MLIR, structured control flow is modeled using standard operations, so support for them is easily incorporated into the interpreter loop. However, because operations can contain blocks which in turn contain operations again, the interpreter extracts the core loop into a separate block evaluation function. This evaluation function can be called recursively to evaluate nested loops and conditionals.

While our interpreter generally does not treat control flow operations as special, it does have some specific logic for handling early break conditions in loops. During parsing, loops with an `until` condition are desugared into loops with a nested if. For an example, see Listing 5.21 and the corresponding IR in Listing 5.22.

Listing 5.21: A program with a loop break condition.

```
func Until(n: int) -> int {
    for i in 0:100 until n > 10 {
        n = n + 1;
    }
    return n;
}
```

Listing 5.22: IR for a program with a loop break condition.

```
graphalg.func @Until(%arg0: i64) -> i64 {
      %c1_i64 = arith.constant 1 : i64
      %c10_i64 = arith.constant 10 : i64
      %idx1 = index.constant 1
      %c100_i64 = arith.constant 100 : i64
      %c0_i64 = arith.constant 0 : i64
      %0 = index.castu %c0_i64 : i64 to index
      %1 = index.castu %c100_i64 : i64 to index
      %2 = scf.for %arg1 = %0 to %1 step %idx1 iter_args(%arg2 = %arg0) -> (i64) {
            %3 = arith.cmpi sgt, %arg2, %c10_i64 : i64
            %4 = scf.if %3 -> (i64) {
                  scf.yield %arg2 : i64
            } else {
                  %5 = arith.addi %arg2, %c1_i64 : i64
                  scf.yield %5 : i64
            }
            scf.yield %4 : i64
      }
      graphalg.return %2 : i64
}
```

Without any special treatment of such loops, the interpreter would always run the loop for 100 iterations, even if the termination condition holds much earlier. This does not affect correctness in any way, since the loop body returns the iter arg unchanged, but it is detrimental to performance.

To address this issue, after each loop iteration the interpreter checks two things:

1. Are any of the values given in `scf.yield` different from the original iter args?

2. Did any of the executed operations have side effects? For example, most GraphBLAS operations used after bufferization do not return any result, but they do write output to a buffer

If neither of these conditions is met, then the state of the program has not changed during this iteration, so running the loop for more iterations would not have any effect on the program state. The interpreter then terminates the loop early and returns the current iter args as the results of the operation. Our approach works well for loops with a break condition and even generalizes to other cases where unnecessary work can be avoided, but the compiler does not detect this. Below we show an example of a loop (without an `until` condition) that our interpreter can terminate after just one iteration:

Listing 5.23: A loop without an `until` that can be terminated early

```
func Term() -> int {
    n = 0;
    for i in 0:10000000 {
        n = n + 1;
        n = n / 2;
    }
    return n;
}
```

While it is nice to see our implementation generalize to other loops, we have not observed realistic programs where it applies to loops without an `until` clause.

Now that we have discussed control flow operations, we discuss the other operation that requires special support in the interpreter, SwapRefOp.

5.5.2 Implementing SwapRefOp

SwapRefOp is the bufferized equivalent of the SwapOp operation. It takes two buffers as arguments and swaps their contents. A naive implementation of the operation is expensive because it requires copying two potentially very large buffers. Because the interpreter stores all variables in a global table, it can instead scan the table for any references to the two buffers, and update them to point to the other buffer. Swapping all pointers has the same effect as swapping the contents, but if the buffers are large and there are not too many pointers, it is much faster.

This concludes our discussion of the interpreter implementation. In the next section, we consider the tradeoffs between interpretation and JIT compilation and explain why we decided to use an interpreter.

5.5.3 Interpreter vs. JIT compilation

The bufferized IR that our pipeline produces is low-level enough that it could straightforwardly be converted into LLVM IR (with function calls to a GraphBLAS implementation), JIT-compiled and executed. This approach is particularly common in compilers that use MLIR because an LLVM IR dialect and JIT compiler integration are readily available. Unfortunately, compiling to machine code has important downsides:

- Lowering the IR to LLVM IR results in a larger IR, which is more difficult to debug.
- Inspecting the program state mid-execution is challenging. Unlike C++ code compiled ahead of time, debug symbols are unavailable.
- The LLVM JIT compiler is known to have significant startup time[58], which adds to the query execution time.

Instead of a full compilation pipeline, we choose to execute the bufferized IR in a simple interpreter. This approach requires less code to implement, is easier to debug, and has low startup time. Running the program in an interpreter is somewhat slower than executing native code, but we expect that for most programs, performing the linear algebra operations will dominate the execution time. These operations will be executed by the GraphBLAS library regardless, so we expect the interpreter to have negligible runtime overhead. We validated this hypothesis in Chapter 7. However, before describing the evaluation in detail, we first cover the operators backend in the next chapter.

Chapter 6

Compilation to Operators

The operators backend compiles graphalg programs into IPR, the internal query IR used in AvantGraph. After compilation, the algorithm becomes regular IPR like any other query, so it relies on the AvantGraph query planner, execution plan generator and execution engine to run the algorithm. The operators backend is therefore tightly coupled to AvantGraph, which offers many advantages:

- Most optimization is taken care of by the query planner, so we do not need to implement this for most of the IPR we generate. Exceptions are cases where we need to introduce new expressions to IPR, in which case there are no existing optimization rules, but these optimizations are typically also useful in regular queries.
- Because both query and embedded algorithms use the same representation, they can be optimized holistically by the query planner. This enables optimizations that are not possible on the GraphBLAS backend.
- Use of the GraphBLAS backend requires that all data is loaded into a GraphBLAS-native graph storage format before it can be processed. The operators backend can execute directly on AvantGraph's native format, eliminating the cost of performing this conversion.

In Figure 2.2 we show an overview of the compiler pipeline.

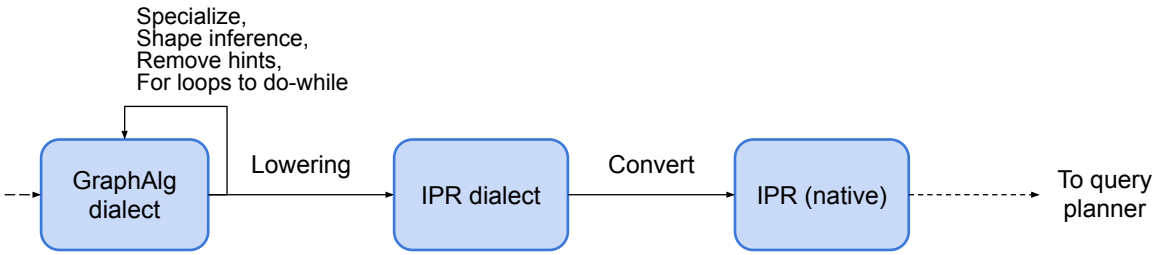


Figure 6.1: Pipeline of the Operators backend.

In the next sections, we explain each pass of the pipeline in more detail. The first is specialization (Section 6.1), where we propagate constants from the query into the algorithm IR. This pass also fixes the dimensions of the input matrices, which are then used in the shape inference pass (Section 6.2) to determine the shape of all matrix types in the algorithm. The operators backend does not use the memory management hints available in graphalg, so we remove them to simplify the IR (Section 6.3). Because of the push-based evaluation model in the operators backend, for loops with break conditions are difficult to implement. To simplify

the runtime, we transform them into do-while loops, such that the break condition can be evaluated in parallel with the loop body expressions (Section 6.4).

Once the IR has been simplified as much as possible and all dimensions are known statically, it is ready to be lowered into the IPR dialect (Section 6.5). The IPR dialect is designed to resemble AvantGraph's native IPR, so any IR expressed in it can be directly converted into IPR (Section 6.6). With the conversion complete, the generated IPR replaces the original algorithm invocation in the query, and the modified query is passed on to the AvantGraph query planner for optimization.

To make IPR expressive enough to support the graphalg language we extend it with additional types of expressions, including query planner support and implementations for these operators in the execution engine. The most important of these is support for loops, whose runtime implementation we discuss in Section 6.7. While IPR provides the necessary primitives to express matrix multiplication, we also add a dedicated semiring operator that offers higher performance. Its design is covered in Section 6.8.

Next to expressions dedicated to graph algorithms, we add apply, aggregate and constant expressions to IPR (see Section 6.9). These are generic operators that are also useful for regular queries but were not yet implemented in AvantGraph.

With the described modifications, IPR is powerful enough to express most graphalg programs. In Section 9.3 we discuss the remaining limitations in AvantGraph that prevent full graphalg support.

6.1 Specialization

When calling a graphalg function in IPR, the parameters of the function may be bound to arbitrary IPR expressions. Often some of these parameters will be bound to constant values, or to expressions of which some properties are known to be constant. For example, consider the following IPR query with an embedded graphalg algorithm:

Listing 6.1: An IPR query passing a constant value to a graphalg function.

```
call(
    "
        func Example(n:int) -> int {
            return n + 1;
        }
    ",
    (%val) = "Example" (101)
)
```

The argument specialization pass propagates such constant values into the function body, replacing all uses of an argument value with the constant. For the function embedded in Listing 6.1, this results in a simplification of the IR:

Listing 6.2: IR before and after specialization

```
// Original IR
graphalg.func @Example(%arg0: i64) -> i64 {
    %c1_i64 = arith.constant 1 : i64
    %0 = arith.addi %arg0, %c1_i64 : i64
    graphalg.return %0 : i64
}

// Specialized
graphalg.func @Example(%arg0: i64) -> i64 {
```

```
    %c102_i64 = arith.constant 102 : i64
    graphalg.return %c102_i64 : i64
}
```

Next to specializing for literal constants, the specialization pass also propagates the dimensions of matrices. If, for example, we run a triangle counting algorithm on the graph, we pass it as an argument to the call expression:

Listing 6.3: Calling TriangleCount on the graph

```
call(
    "
        func TriangleCount(graph: Matrix<bool>) -> int { ... }
    ",
    (%val) = "TriangleCount" (
        matrix(
            projection(
                access(%0, "friend"), {
                    %row = src(%0),
                    %col = trg(%0),
                    %val = %0.weight,
                }
            ),
            %row, %col, %val),
    ),
)
```

AvantGraph keeps statistics about the graph, including the number of vertices, so at the time the algorithm is compiled it fixes the dimensions of the `graph` parameter. For a graph with 42 vertices, the specialization pass applies the following transformation:

Listing 6.4: TriangleCount IR before and after specialization.

```
// Original IR
graphalg.func @TriangleCount(%arg0: tensor< ?x ?xi1>) -> i64 { ... }
// Specialized
graphalg.func @TriangleCount(%arg0: tensor<42x42xi1>) -> i64 { ... }
```

With the dimensions of the input matrices known, we can apply the next pass, shape inference, to infer dimensions outputs of operations that depend on the arguments.

6.2 Shape Inference

Graphalg programs do not statically define the sizes of matrices, but to lower some operations into IPR, the dimensions of the inputs must be known. The shape inference pass is a collection of rewrite rules that propagate known matrix dimensions through operations. For most operations, the shape of their output is the same as the shape of the input. Exceptions to this are the semiring, transpose and reduce_rows operations, for which we give the rewrite rules in Listing 6.5.

Listing 6.5: Shape inference rules for semiring, transpose and reduce_rows. Identifiers with capital letters represent placeholders for IR fragments. `%a` and `%b` are arbitrary MLIR values.

```
semiring (%a:tensor<Rx_xT0) SR (%b:tensor<_xCxT1>) -> tensor<?x?xT3> =>
semiring (%a:tensor<Rx_xT0) SR (%b:tensor<_xCxT1>) -> tensor<RxCxT3>

transpose $a:tensor<RxCxT> -> tensor<?x?xT> =>
```

```
transpose $a:tensor<RxCxT> -> tensor<CxRxT>

reduce_rows OP %a:tensor<RxCxT> -> tensor<?xT> =>
reduce_rows OP %a:tensor<RxCxT> -> tensor<RxT>
```

6.3 Remove Hints

After shape inference, we run a pass to remove memory management hints. Three operations are removed from IR, and replaced with a simpler expression:

- `swap` operations are deleted, and references are updated to point directly to the original values.
- `clear` operations are removed by replacing them with a new matrix allocation.
- `assign` operations that do not specify a range do not take any values from `base` and effectively copy `value`. They are replaced with a direct reference to `value`.

In Listing 6.6 we give a small graphalg program that can be simplified by this pass. The corresponding IR before and after the transformation is given in Listing 6.7.

Listing 6.6: A graphalg program with memory management hints.

```
func Example(a:Vector<int>,
             b:Vector<int>,
             c:Vector<int>) -> Vector<int> {
    swap a b;
    c = b;
    return c;
}
```

Listing 6.7: IR simplification using the memory management hints removal pass.

```
// Original IR
graphalg.func @Example(%arg0: tensor<?xi64>,
                       %arg1: tensor<?xi64>,
                       %arg2: tensor<?xi64>) -> tensor<?xi64> {
    %0:2 = graphalg.swap %arg0 %arg1 : tensor<?xi64>
    %1 = graphalg.assign %arg2 : tensor<?xi64> =
        %0#1 : tensor<?xi64> -> tensor<?xi64>
    graphalg.return %1 : tensor<?xi64>
}

// Hints removed
graphalg.func @Example(%arg0: tensor<?xi64>,
                       %arg1: tensor<?xi64>,
                       %arg2: tensor<?xi64>) -> tensor<?xi64> {
    graphalg.return %arg0 : tensor<?xi64>
}
```

6.4 For to Do-while loops

There is one step remaining until we can lower to IPR: all for loops must be converted into do-while loops. For loops with break conditions have semantics that are difficult to map to

the push-based, bottom-up evaluation strategy in AvantGraph. Consider for example the following loop:

Listing 6.8: A for loop with a break condition

```
func ForSimple(n:int) -> int {
    for k in 1:10 until n >= 256 {
        n = n + n;
    }
    return n;
}
```

Executing this loop in the AvantGraph runtime is problematic because before we can evaluate the expression in the loop body, the loop condition must be evaluated. However, the loop condition and body both depend on the iter args, so if we provide the necessary inputs to the condition expression, the body will already start to be evaluated. While this issue could be solved by speculatively executing loop bodies or stalling the execution of certain operators temporarily, this would add significant complexity to the runtime. Instead, we apply a transformation to the IR that preserves the original semantics of the program, yet allows evaluation of loop body and condition in parallel.

This transformation is done in two steps. First, we transform the for loop into a while loop with a single condition. Using Listing 6.8 as an example, after this first transformation step the program becomes:

Listing 6.9: A for loop after conversion to a while loop.

```
func ForSimple(n:int) -> int {
    k = 1;
    while k < 10 && n < 256 {
        n = n + n;
        k = k + 1;
    }
    return n;
}
```

This first step makes the iteration process explicit and folds the range check into the condition expression, simplifying the loop. The condition however still needs to be evaluated before we can run the loop body. To avoid this, we peel off the first condition check and place it before the loop to run directly on the initial arguments. Now if this initial check passes, we can enter the loop and immediately execute the body. The condition only needs to be re-evaluated *after* each iteration, so it naturally depends on the results of the loop body. We have effectively converted the for loop into a do-while loop:

Listing 6.10: A for loop after conversion to a do-while loop.

```
func ForSimple(n:int) -> int {
    k = 1;
    if n < 256 {
        do {
            n = n + n;
            k = k + 1;
        } while k < 10 && n < 256
    }
    return n;
}
```

In this particular example, the condition can only be evaluated after the body has been fully evaluated, but in more complex programs with many loop-carried dependencies, the condition can often be evaluated in parallel with the body.

An apparent issue with this transformation is that it introduces if statements into programs, which the operators backend does not yet have support for. Fortunately, many programs have constant initial arguments to loops, particularly after argument specialization, so the compiler can usually determine statically that the loop condition will hold on the first iteration, and omit the if statement. The program from Listing 6.10 is no exception: if we specialize it with $n = 1$, the if condition trivially holds, and we obtain the following program:

Listing 6.11: A for loop after conversion to a do-while loop, specialized for $n = 1$.

```
func ForSimple(n:int) -> int {
    n = 1; // specialized
    k = 1;
    do {
        n = n + n;
        k = k + 1;
    } while k < 10 && n < 256
    return n;
}
```

After converting for loops into do-while loops, the graphalg IR reaches its final form. In the next step, it is lowered into the IPR dialect.

6.5 Lowering to IPR

The lowering pass is by far the most involved in the pipeline: all operations in the IR are rewritten to operations from the IPR dialect. The full rewrite is necessary because expressions in IPR return a stream of tuples, so all operations must be adapted to return values of this type. A tuple stream is modeled with the custom MLIR type `ipr.tuple_stream<columns>`. An instance of the tuple steam type has one or more columns, each identified with a globally unique id. Every column also has an associated type for the elements that are returned from that column. Before going into the details of how each operation in the graphalg dialect is lowered into IPR, we first discuss how types are converted into these tuple streams.

6.5.1 Graphalg Types to Tuple Streams

Scalar values (types `i1`, `i64`, `f64` and `index`) map to a stream with a single column (and a single value, but this is not encoded in the type). For a graphalg operation with return type `i64`, after lowering the return type thus becomes a type like `ipr.tuple_stream<42:i64>`, where `42` can be replaced with any number that is not already used in the IR for another stream.

Vectors are encoded as a stream where each tuple contains the row number and the value for that entry. A type such as `tensor<10xi1>` becomes `ipr.tuple_stream<43:index,44:i1>` (information about the dimensions of the vector is lost after lowering).

For matrices, the encoding resembles that of vectors but includes a column index. The matrix type `tensor<100x100xf64>` maps to `ipr.tuple_stream<45:index,46:index,47:f64>`. Next, we discuss how the individual operations are mapped to IPR.

6.5.2 SelectOp

SelectOp has a direct analog in relational algebra, the σ operator, which makes the lowering trivial. For example, the graphalg expression `select(==, A, 0)` is equivalent to $\sigma_{A.val=0}(A)$. Besides comparison operators which compare each entry of the matrix to some value, graphalg

also supports the `tril` and `triu` predicates for selecting elements from a lower or upper triangle of the matrix, respectively. The predicates are defined as:

$$tril(r, c, y) = c \leqslant (r + y)$$
$$triu(r, c, y) = c \geqslant (r + y)$$

Where y is an additional parameter that specifies above/below which diagonal entries are filtered. AvantGraph only supports comparisons between slots and constant values in selection predicates, so we cannot lower `tril` and `triu` predicates for all values of y. Luckily the programs where we have encountered these predicates always have $y \in \{-1, 0, 1\}$, in which case simple comparisons of slots are sufficient. For example:

$$tril(r, c, -1) = c < r$$
$$triu(r, c, 1) = c > r$$

6.5.3 TransposeOp

Whereas in GraphBLAS a transpose operation is potentially expensive because of the data format, the operators backend does not define an order over the elements of a tuple stream, so a projection to swap the row and column suffices. The graphalg operation `A.T` maps to $\Pi_{row=A.col,col=A.row,val=A.val}(A)$. AvantGraph removes all projections when generating an execution plan, so transposing a matrix comes at no runtime cost.

6.5.4 SemiringOp

The lowering of SemiringOp is crucial for the operators backend because it is typically the most computationally expensive operation in an algorithm. In this section, we discuss a strategy for lowering matrix multiplication into standard relational algebra operators. An alternative specialized operation with higher performance and integrated masked assignment is described in Section 6.8.

A conversion of matrix multiplication into relational algebra can be derived from the definition:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

To compute this in relational algebra we must:

1. Make tuples with each A_{ik} and B_{kj}. We do this using a join: $A \bowtie_{A.col=B.row} B$.

2. Multiply the two entries together. We use the binary function application operator described in Section 6.9.

3. For each entry of the result, sum the results of the multiplication. In relational algebra, this is a standard aggregation with grouping. The aggregation operator is also described in more detail Section 6.9.

The apply and aggregate operators are generic over the function they apply, so we use the same strategy to perform matrix multiplication over arbitrary semirings. In Figure 6.2 we give an illustration of generalized matrix multiplication lowered to relational algebra.

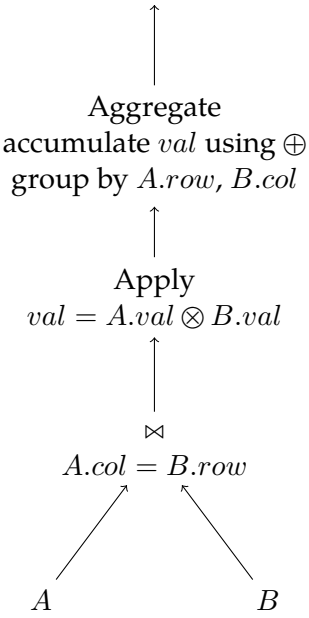


Figure 6.2: Matrix multiplication $A(\oplus.\otimes)B$ lowered to relational algebra.

6.5.5 MaskOp

The lowering of a masked assignment depends on the value of `base` and the descriptor flags. The simplest case is `C<M> = A` where `C` is a new or empty matrix. In this case C_{ij} is defined as:

$$C_{ij} = A_{ij} \quad \text{if } M_{ij} = true$$

Expressed in relational algebra, A and M are joined on their row and column. A selection predicate filters elements from M that do not have the value $true$. Finally, a projection removes the additional columns from M again. The expression tree is given in Figure 6.3.

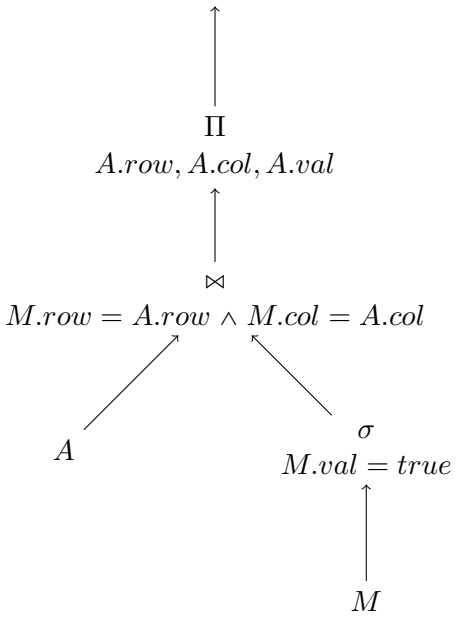


Figure 6.3: Masked assignment in relational algebra.

If the `structural` flag is set, the selection operators over M can be omitted, since only the presence of entries is significant. MaskOps with a base or complemented mask are not

yet supported. If the MaskOp can be fused to a SemiringOp though, this functionality is implemented directly by the semiring operator (Section 6.8).

6.5.6 ReduceOp & ReduceRowsOp

Both reduction operators can be lowered directly to an aggregation operator (see Section 6.9). For ReduceOp no grouping keys are needed, while for ReduceRowsOp the row is a grouping key.

6.5.7 AssignOp

AssignOp serves two purposes:

1. Assign a scalar value to multiple entries of a 'base' matrix.

2. Assign a matrix to a submatrix of a 'base' matrix.

Only the assignment of scalar values has been implemented, we discuss its implementation below. Assigning a submatrix can be expressed in relational algebra, but we have not found a need for this functionality, so we have not implemented it.

First, we look at an assignment of the form `V[a:b] = n`, where `V` is an empty vector. The result of this statement should be a vector with value `n` for all positions between `a` (inclusive) and `b` (exclusive). In relational algebra, this vector is generated by creating the constant range $a, a+1, a+2, \ldots, b-1$, and computing a cartesian product with the value n, as shown in Figure 6.4.

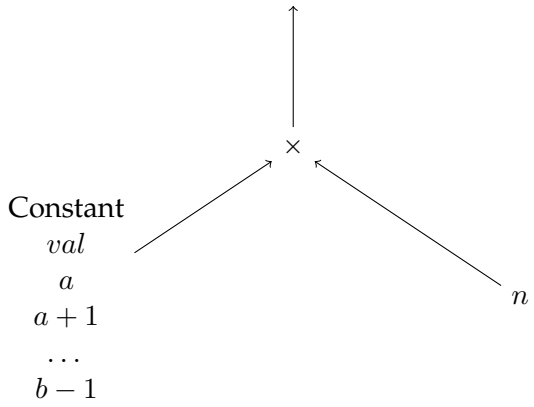


Figure 6.4: Assigning a scalar value to a range.

The value n does not have to be constant. It can be defined by an arbitrarily complex expression that AvantGraph can dynamically evaluate. While Figure 6.4 may suggest that n is passed to the join as a scalar value, such a type does not exist in IPR. Instead, n is a stream with a single column and one tuple.

The range `a:b` must be constant at algorithm compilation time. `a` and `b` can be literals in the algorithm, or they can be determined by an expression that the compiler can fold to a constant. Expressions for `a` and `b` may also depend on the dimensions of the graph or any matrix derived from it, so long as the dimensions can be fixed by the shape inference pass.

A special case of this is the 'all' range. If instead the assignment were of the form `V[:] = n`, the range depends on the number of rows in `V`. So long as shape inference can determine the dimensions of `V`, this statement compiles. It translates to the same relational algebra expression, only with $a = 0$ and $b = V.nrows$.

If a base matrix is present, a selection operator must first be applied to clear existing elements from the range, after which we compute a union with the expression shown in Figure 6.4.

6.5.8 Casting

Graphalg supports casting of scalar types, e.g. `real(42)` converts the integer 42 into a floating point number 42.0. While IPR is not explicitly typed, the inputs to binary operators must have matching types to be valid at runtime, so we insert cast operators into IPR just like we do in graphalg IR. Since in IPR scalars are tuple streams with a single column, we can implement casts as unary functions for the apply operator (see Section 6.9).

6.5.9 For loops

Loops in graphalg are lowered to a for loop operator specifically designed for this purpose. We discuss its implementation in detail in Section 6.7. The IPR loop operator is more restricted in functionality than the for loop construct in graphalg. The following properties are important for the translation:

- IPR loop bounds are over constant integers, whereas graphalg allows these to be dynamically computed. There is currently no workaround for dynamic bounds. If constant bounds cannot be determined for a loop, compilation fails. In practice, we have found most loop bounds to be constants, especially after running the specialization and shape inference passes. We note that this limitation on IPR loops is imposed merely to simplify the implementation: a future extension may lift this requirement and allow for a wider range of programs to be accepted by the backend.

- In graphalg IR, loops can return any number of results. However, because IPR requires a tree structure, only a single result can be returned from an IPR loop. To convert loops that return multiple results, a separate IPR loop is created for each result. We have found loops with multiple results to be uncommon, but when necessary this conversion does lead to inefficient query plans. For more details, see Section 9.3.

6.5.10 Scalar Arithmetic and Compare operations

Graphalg IR can contain many different operations that accept two scalar values as input and return another scalar as output. This includes arithmetic operations (AddIOp, SubFOp, etc.) but also comparison operations (CmpIOp, CmpFOp). Because of the structural similarities, all these operations are lowered using the same strategy:

1. The two scalar inputs, now tuple streams, are combined using a join operation. Because both streams produce exactly one value, no join condition is needed.

2. An apply operator (see Section 6.9) evaluates the desired function for the two inputs.

3. Using a projection, the original two input columns are dropped, leaving only the output column.

As an example, in Figure 6.5 we give the IPR for the graphalg expression $1 + 2$.

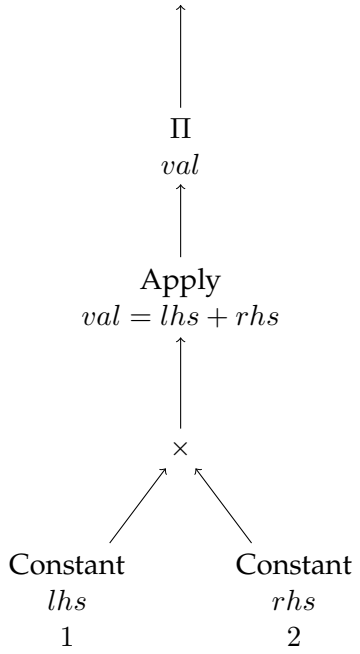


Figure 6.5: Scalar addition in IPR.

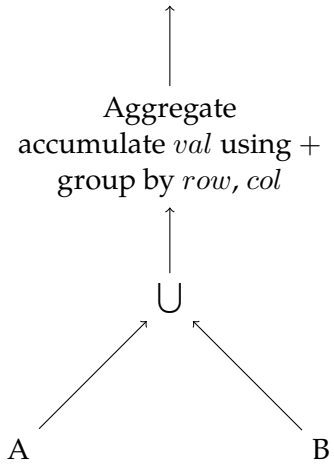


Figure 6.6: Accumulation in IPR.

6.5.11 AccumOp

Accumulate operations are lowered into IPR by first creating the union of the `base` and `value` tuple streams, grouping the results by row and column, and aggregating results using the accumulation function. For a graphalg statement `A += B`, the generated IPR is given in Figure 6.6.

This approach works only for operators that are commutative since any ordering of elements in `A` vs. those in `B` is lost. We support `A -= B` as a special case by applying a unary negation to the elements of `B` first, after which we can aggregate the results using $+$.

6.5.12 ApplyUnaryOp and ApplyBinaryOp

Unary/binary function application is lowered to an IPR apply expression (see Section 6.9). For ApplyBinaryOp, where the second input is a scalar, we have the additional requirement that it is a constant since inputs in IPR apply must be slots or constants. This limitation could

be lifted with an additional join operator, but we have not needed it in any programs, so it is not implemented. As an example, in Figure 6.7 we give the IPR for `apply(abs, A)`.

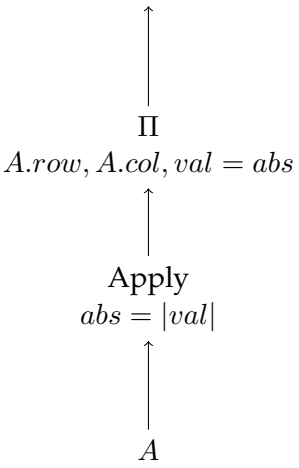


Figure 6.7: Unary function application in IPR.

6.5.13 ElementWiseOp

To lower ElementWiseOp we use one of two strategies, depending on the join mode. For the intersection mode, we combine the two matrices using a join operator, then apply the function with an IPR apply expression. An example with the division operator is given in Figure 6.8.

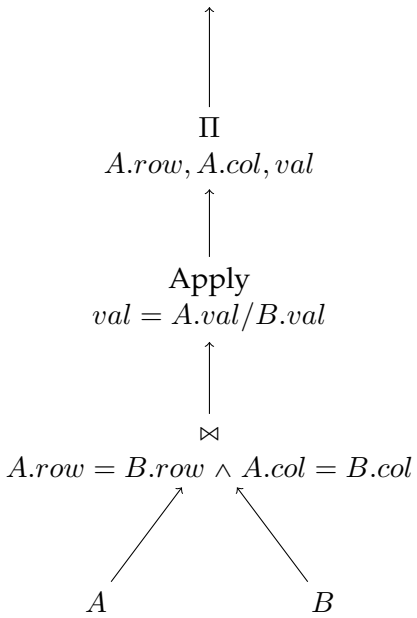


Figure 6.8: Element-wise division in IPR (intersection mode).

In union mode, we also need to preserve elements that are only present in one of the two inputs, so we cannot use an inner join. As established in Section 3.3, an ElementWiseOp in union mode is identical to an accumulate operation, so the expression `add(+, A, B)` lowers to the IPR shown previously in Figure 6.6.

6.5.14 NvalsOp

The NvalsOp counts the number of values present in a matrix. In our tuple stream representation, this is equivalent to counting the number of tuples. We implement this by joining every tuple with the constant 1, then accumulating the 1 values to obtain the count. If there are no tuples in the input stream we should return a single tuple with value 0 instead of outputting nothing at all. To achieve this, we also feed an initial 0 value into the aggregation using a union operator. The IPR generated for an NvalsOp is given in Figure 6.9.

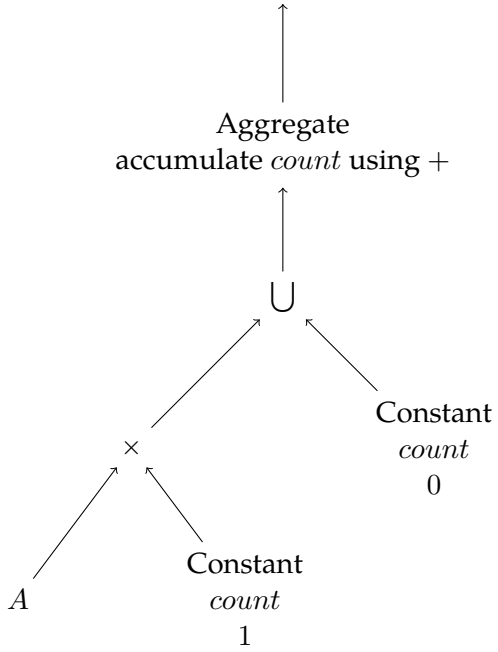


Figure 6.9: NvalsOp in IPR.

6.6 Conversion to Native IPR

In the previous section, we discussed how to convert graphalg operations into the IPR in the abstract, with no regard for how the IR can 'escape' MLIR and be converted into native IPR data structures as they are used by AvantGraph. This section covers those details.

6.6.1 Operations to IPR Expressions

The IPR dialect is designed faithfully mimic the expressions available in IPR. Each operation in the IR is mapped directly to the equivalent IPR expression. Unlike the IPR dialect, native IPR is untyped, so all type information is lost in the translation process. The slot numbers encoded in the types op expressions however are preserved as properties of the IPR expressions. The translation is done recursively from the return statement of the called function. For operations that have other operations as an input (e.g. a join operation with child inputs), we first convert the inputs, after which the parent expression can be constructed.

6.6.2 Multiple References to Operation Results

It is common for the generated IR to contain operations with a result that is used as an input to multiple later operations. In MLIR using the same value twice requires no special treatment, but in IPR we must take special care to avoid evaluating such an expression multiple

times. If in IPR we include the same expression multiple times in the query plan, the planner constructs separate instances of it in the execution plan, which is typically inefficient. To ensure that an expression is evaluated once no matter how many times it is referenced in the query, IPR supports *subexpressions*. Subexpressions are named expressions stored alongside the main query expression, which can be referenced any number of times by the main query by their name. In the final execution plan, each subexpression is instantiated once, and their outputs are broadcast to all operators that refer to it. When converting operations into IPR expressions, the compiler checks how many times the result is used. If it is used more than once, the expression is placed into a subexpression. While it would be safe to conservatively place every converted operation into a subexpression, this creates a bloated and difficult-to-optimize query plan, so if the compiler detects that a result is used exactly once, no subexpression is generated.

6.6.3 Loop arguments

Like multiple references, loop body arguments are another construct that is trivial to represent in MLIR, but requires special consideration in IPR. IPR has no concept of regions or blocks, so instead we create a special *iter arg* expression. An iter arg is similar to a subexpression reference, but instead of referring to an expression defined outside of the query, it references a result of a previous loop iteration. When a loop iteration completes, the loop expression has received all inputs from the body. If the loop condition still holds, the received blocks are sent back to the corresponding iter args, and a new iteration begins. In Figure 6.10 we illustrate the interaction between loop and iter arg expressions in a query plan.

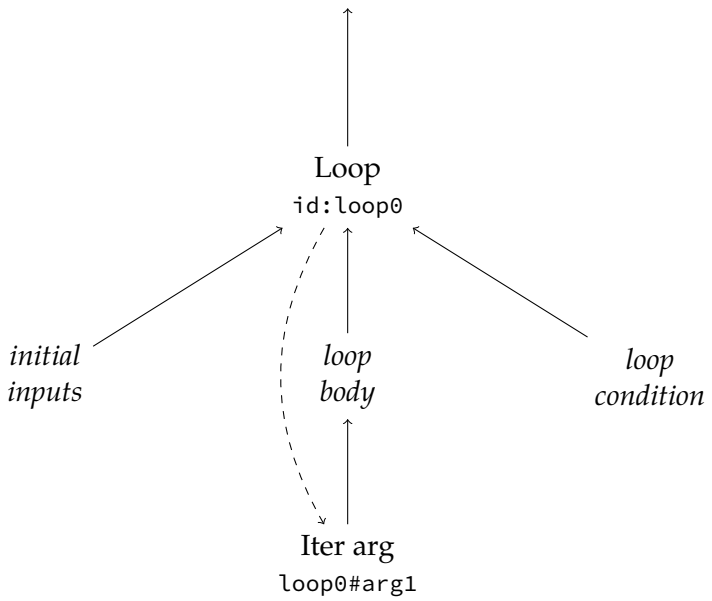


Figure 6.10: An iter arg expression in a query plan. The relation between the loop expression and iter arg drawn with a dashed arrow is implicit, and not part of the IPR structure.

The loop and iter arg expressions are new constructs in IPR that we add specifically to support graph algorithms. In the next section, we discuss how they are implemented.

6.7 Implementing a Loop Operator

The IPR loop expression has the following inputs:

- Initial arguments: child expressions that produce the initial values for the loop-carried variables.
- Loop body expressions: child expressions that produce the next values for the loop-carried variables. Together they represent a single loop iteration. They may depend on iter args, which provide the current values of loop-carried variables. Loop body expressions must produce the tuples with the same slots as the corresponding initial arguments.
- Loop condition expression: a child expression that produces a single tuple. The tuple must contain a boolean (*bit* is the terminology used in the runtime) column indicating if the loop should terminate (`false`) or run for another iteration (`true`).

When converting the query plan into an execution plan, most of this structure is preserved: the loop expression becomes a loop operator, and the iter arg expressions become iter arg operators. One important difference however is that initial arguments are not inputs to the loop operators, but instead, they are directly connected to the corresponding iter arg. The blocks of the initial arguments need to be forwarded to the iter args to run the first iteration, and connecting them directly makes this easy. Once the loop body expressions have produced all their output blocks and the loop operator has buffered them, the loop operator moves the buffered blocks to the iter args. The loop operators then *resets* the iter arg operators to prepare them for a new iteration. This also triggers a reset of all operators whose output is affected by the iter args, which amounts to all operators in the loop body. After all loop body operators have been reset, the iter args start to push their received blocks, triggering the start of a new loop iteration.

We introduce *reset* functionality into the runtime specifically to support iteration. In the next section, we discuss it in more detail.

6.7.1 Resetting operators

When the iter arg operators are ready to push their new blocks, we expect the loop body operators to process them and push the output toward the loop operator. However, after the first iteration completes, those operators have already produced output. As a result, they have been marked by the runtime as *completed*, indicating they will not produce new tuples. Completed operators are not monitored by the runtime for new inputs to process, stalling the query. Furthermore, some operators are stateful, and may still be storing data from the previous iteration. Therefore, to run a new iteration, the completion status of the loop body operators must be reset, and their internal state returned to initial conditions. To achieve this we extend the runtime with *reset* functionality: an operator can request the query coordinator to reset an operator x in the execution plan. The coordinator then clears the completion status of x and clears its internal state. To ensure that the new blocks x will output can be processed by its parents, we also reset any operators that have x as an input. A complication occurs for operators that have multiple inputs, such as a join. If only one of its inputs is reset, the entire join operation must be performed again. Not only do we need to reset the parent of the join, but we also need to reset *all* inputs to the join, even those that do not depend on an iter arg operator. We give the full reset algorithm in Listing 6.12.

Listing 6.12: Operator reset algorithm.

```
// To reset a loop, call on all iter args.
// Initialize resetOperators with the index of the loop operator.
void reset(OperatorIdx idx, set<OperatorIdx> resetOperators) {
    if (resetOperators.contains(idx)) {
        return;
```

```
    }
    resetOperators.insert(idx);
    operators[idx].completed = false;

    // Do not reset inputs of iter arg, they only need
    // to be evaluated at the first iteration.
    if (operators[idx].type != FOR_ITER_ARG) {
        for (OperatorIdx input: inputs(idx)) {
            reset(input, resetOperators);
        }
    }
    for (OperatorIdx output: outputs(idx)) {
        reset(output, resetOperators);
    }
}
```

6.7.2 Build and Destroy Operators

The presented algorithm may appear to reset nearly the entire query plan at every iteration, but this is rarely the case. An important reason for this is that IPR join expressions are converted to multiple operators in the execution plan, and those operators do not have a simple input-output relation. If we take as an example a join of A and B, which the query planner has determined should be computed using a hash join with A on the probe side, and B on the build side. The corresponding execution plan is given in Figure 6.11.

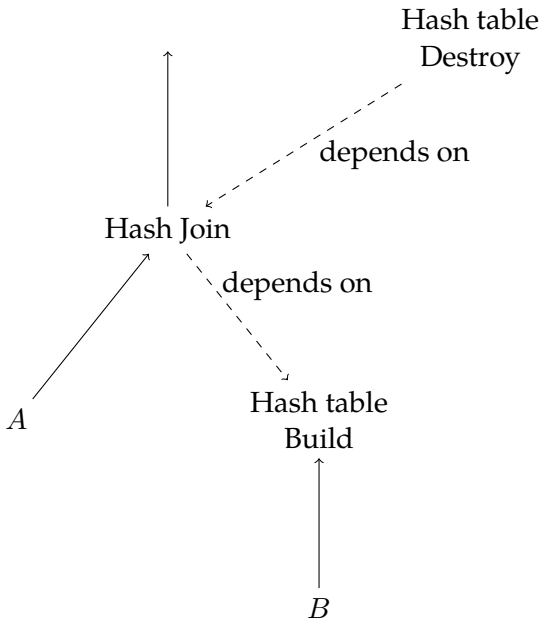


Figure 6.11: Execution plan for a hash join.

Three operators have been created for the join expression:

- A build operator to create a hash table of B.
- A join operator that receives tuples of A and probes the constructed hash table. The join and build operators are related by a *depends on* edge, meaning the join operator will only start processing once the build operator has completed.

- A destroy operator that waits until the join is completed (via another *depends on* edge), and frees the memory allocated for the hash table.

If this plan fragment were inside a loop, and A depends on an iter arg but B does not, then the hash table does not need to be rebuilt for the next iteration. This is nicely captured by the plan because the reset function propagates only over input-output relations, not *depends on* edges. We do need to be careful though where the *depends on* edge from the destroy operator points to. If the join is part of a loop body and the build operator does not depend on an iter arg, the destroy operator should run after the loop operator completes instead of after the join, to preserve the table across iteration. Our implementation performs this analysis once in the execution plan generator and places the *depends on* edge accordingly.

6.8 Designing a Semiring Operator

Graphalg semiring operations can be translated into the standard relational algebra operations join, apply and aggregate. Because semiring operations represent graph traversal steps, they are fundamental to nearly all algorithms, and hence the performance and functionality of this operation are crucial. In our experiments, we found two important problems with the transformation of semiring operations into standard relational algebra, stemming from limitations of AvantGraph:

- AvantGraph does not implement merge join, so a hash join is used instead. The larger size of the join is typically the graph (or a large subset of it), leading to a very expensive build phase that loads the entire graph into a hash table. The other side of the join is smaller (on the order of the number of vertices in the graph), but this is still sufficiently large that many probes into the table are needed. The result is high memory consumption and low performance.

- complemented masked assignment with the result of a semiring, such as `r<!pi, struct> = q (any.secondi) graph`, is a common pattern in graph algorithms. For example, in breadth-first search, this is used to avoid revisiting the same vertex multiple times. Complemented masks can be expressed in relational algebra with an *anti-join*. An anti-join filters out tuples from its *source* relation if join to a tuple in the *filter* relation. Unfortunately, this type of join is not supported in AvantGraph, and implementing it would take significant engineering effort.

To solve these issues, we implement a dedicated semiring expression in IPR with built-in support for complemented masks. We also implement a custom lowering to query operators that are optimized for matrix operations. Depending on the rank of the input matrices (graphalg support matrix-matrix, vector-matrix and matrix-vector) and whether or not a (complemented) mask is present, we lower to different query operators. We focus on two common cases where we get the most benefit: Masked matrix-matrix with right transpose, for which we show an execution plan in Figure 6.12, and vector-matrix with an optional complemented mask, shown in Figure 6.13.

6.8.1 Building CSRs

The two execution plans are quite different, but they do share a 'CSR Matrix Build' operator. Instead of building a hash table, the semiring operator we build a graph representation based on the *compressed sparse row* (*CSR*) format [46]. This representation is better optimized for sparse matrices where fast access to rows is required. It is also one of the formats used by GraphBLAS. In Figure 6.14 we show a classical CSR.

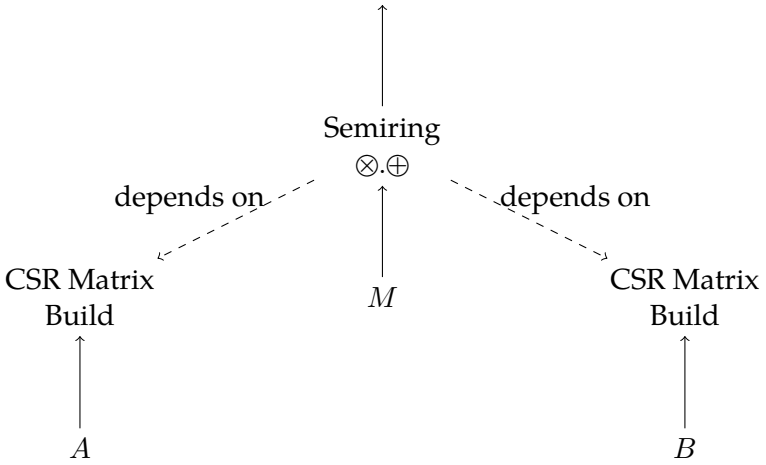


Figure 6.12: Execution plan for masked matrix-matrix multiplication.

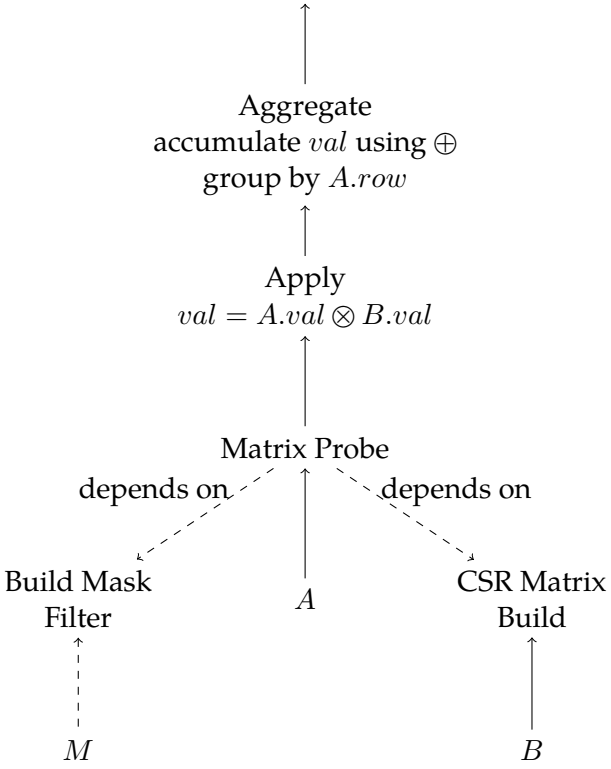


Figure 6.13: Execution plan for Vector-Matrix multiplication with a complemented mask.

The col array stores the column index of each value in val. The entries are sorted by row (and column), so instead of specifying the row number for each value, it is sufficient to store where each row begins and ends. The rowindex array does exactly this: `rowindex[`x`]` stores the offset into col or val where row x begins.

Our format is very similar but builds the CSR directly over the blocks of tuples that it receives from the input to avoid unnecessary copies. Instead of a single offset value, our rowindex stores both the block sequence number and the offset inside the block. We show the same matrix in our format in Figure 6.15. The rowindex is built by scanning the row column once all input blocks have been received.

Building a CSR requires that the matrix entries are sorted by row and column. AvantGraph does not provide this guarantee even for base tables (see Section 9.3). Sorting a large

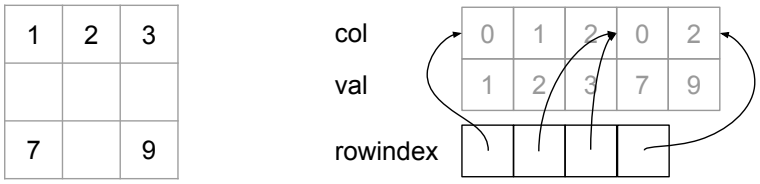


Figure 6.14: A matrix represented in the classical CSR format with flat arrays.

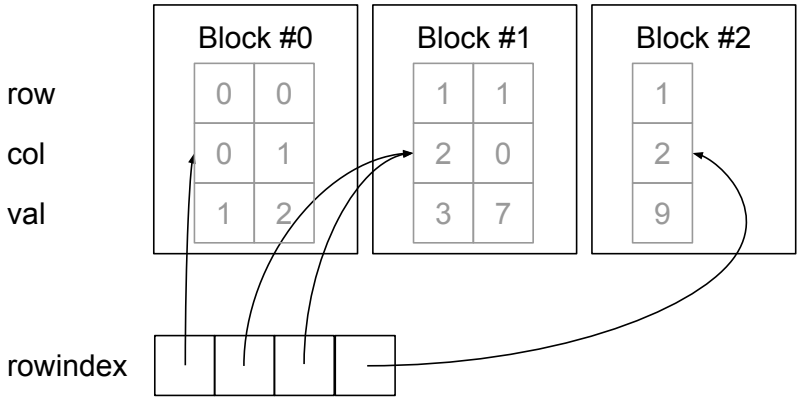


Figure 6.15: Our modified CSR format constructed over blocks.

set of tuples would be prohibitively expensive, so we have made additional modifications to the runtime to keep data (mostly) sorted. Firstly, when loading input graphs, we provide the edges to the loader in sorted order, so that at least the base table data is sorted. This guarantees that blocks are initially created in sorted order from the base data, but the Avant-Graph runtime may process blocks in parallel and does not guarantee that they are delivered in order. While the tuples within a block will remain in sorted order, intermediate operators may cause the blocks themselves to be reordered. To address this problem we tag blocks originating from a base table with a sequence number and propagate it where possible (e.g. through selection operators). The CSR build operator uses this sequence number to sort the blocks, recreating the original sorted tuples.

We currently perform the sorting and rowindex building on a single thread, which is a potential performance bottleneck. In practice, however, we have not found it to have a significant effect on the runtime of any algorithm, even on a machine with 46 cores.

6.8.2 Matrix-Matrix

For matrix multiplication where the right side is transposed, entries of the output matrix are computed as:

$$C_{r,c} = A_{r,0} \otimes B_{c,0} \oplus A_{r,1} \otimes B_{c,1} \oplus \cdots$$

Because this is a masked matrix multiplication, the output should only contain $C_{r,c}$ if $M_{r,c}$ is present. We implement this by first building CSRs for A and B, then we stream in M and produce results as we read entries of M. For each tuple (r, c, v) of M, we load row r from A and row c from B. We multiply elements of the two rows in element-wise fashion using $\otimes$, then aggregate the result using $\oplus$. Our method requires only row-wise access to A and B, which is very efficient in the CSR format.

6.8.3 Vector-Matrix

Our approach for vector-matrix is different from the matrix-matrix strategy, and instead resembles the relational algebra approach. The key difference though is that instead of probing a hash table, we now probe a CSR matrix. Probing the CSR does not require hashing or testing for collisions, and is therefore much faster. If the optional complemented mask is specified, a filter is built before the probing starts. The probe operator checks if entries are also present in the filter, and if so it skips them. The filter is implemented as a bitvector.

We found that in the breadth-first search algorithm, building the mask filter can take up as much as 1/3 of the total execution time, so we have implemented additional optimizations to make it faster. The simplest way to populate the filter is to loop over the entries of M and write to the filter once for every entry. The X86-64 CPU that we run our experiments on has a word size of 64 bits, so writes of a smaller size suffer a performance penalty, and in general memory on the X86-64 architecture is byte rather than bit-addressable. To try and make the most of our 64 bit writes, we do not write bits directly to the filter, but instead we keep one 64-bit chunk in a register. So long as consecutive entries of M all map to the same chunk, we can simply set an extra bit in the register. Only once an entry appears that falls outside the current chunk do we write it to the filter, and place the chunk for the new entry in the register instead. We have also experimented with partitioning the entries of blocks before looping over them to avoid switching chunks, but we found that the reduced amount of writes did not offset the cost of partitioning.

6.9 Other Operators Added in AvantGraph

Next to the loop and semiring operators, which are designed specifically for use in graphalg, we extend AvantGraph with three generic operators that are also useful for regular graph queries:

- A constant operator that allows embedding small constant tables directly in queries.
- An apply operator that applies a unary or binary function.
- An aggregation operator that aggregates values of one column, optionally grouped by one or more key columns.

6.9.1 Constant operator

The constant operator defines a small table that is inlined into the query plan. It is useful for representing literals included in a query or algorithm.

To support the translation of large graphalg ranges, we also include bounded range generation into the constant operator. Expanding such a range at compile-time can potentially consume a lot of memory, so in the constant operator, we allow specifying only the endpoints of the range for a particular column. The entries of the range are computed at runtime and produced one block at a time.

6.9.2 Apply operator

The apply operator applies a unary or binary function to certain columns of each tuple. For unary functions, the argument must be a column, whereas for binary functions one of two inputs can also be a constant value. We currently implement the following functions:

- Casting (unary): to vertex id, integer or real.
- Arithmetic (binary): add, subtract, multiply, divide.

- Arithmetic (unary): absolute value, unary negation.
- Compare (binary): $=, \neq, <, \leqslant, >, \geqslant$.
- Logical (binary): logical and.

The apply operator does not modify any of the original columns in the stream but instead adds an extra column containing the result.

6.9.3 Aggregation operator

The aggregation operator takes as input a column to aggregate, a merge function or *monoid* used to aggregate values, and optionally several key columns. The output stream contains only the aggregation and key columns. One tuple for every unique combination of key values in the input stream is returned. If the input stream contains multiple tuples for the same key values, they are combined using the merge function.

The operators backend often inserts aggregation functions to combine two vectors, and it is also used to perform matrix multiplication if the semiring operator is not used. For these purposes, the aggregation operator always uses the row and column index as keys. Because this case is so common, we provide a specialized implementation for it with a hash table optimized for small keys.

This concludes the chapter on the operators backend. Now that we have covered the design of the compiler and its backends, we continue with an evaluation of the system.

Chapter 7

Evaluation

We evaluate our implementation on the design goals stated in Section 3.1:

- **Efficiency**: We compare the execution time of the GraphBLAS backend against LAGraph, a collection of state-of-the-art graph algorithm implementations based on the GraphBLAS library (Section 7.2). We also compare the performance of the GraphBLAS and operators backends based on total execution time (Subsection 7.3.3), graph loading time (Subsection 7.3.4) and peak memory consumption (Subsection 7.3.5). Additionally, we verify that our optimizations are generic enough to apply to new algorithms (Subsection 7.4.1).

- **Ease of programming:** Graphalg is a high-level language that is memory safe and terminating. With built-in support for linear algebra operations, graph algorithms can be expressed concisely, and without explicit parallelism. Many potential users will already be familiar with linear algebra since it is part of the curriculum in many (engineering) degrees. We believe these qualities make graphalg easy to learn and use. However, we recognize that the accessibility of the language is a subjective quality that is difficult to quantify, especially now that the language has no active users. Running a user study to investigate users' experience with graphalg is outside of the scope of this thesis, and left as future work. We do not discuss ease of programming further in this chapter.

- **Safe to run inside a database:** Graphalg is a memory safe language. Furthermore, all loops are bounded, and recursion is forbidden, providing a guarantee that all graphalg programs terminate eventually. Graphalg thus satisfies our definition of safe execution in a database. There may still be bugs in the implementation that break these properties. Extensively testing our implementation for safety violations, however, is out of scope for this work. We do not discuss safety further in this chapter.

- **Expressivity:** Graphalg can express all programs in the GAP benchmark suite. Our current GraphBLAS backend can execute efficient implementations of the Triangle Count (TC), Single-Source Shortest Paths (SSSP), PageRank (PR) and Breadth-First Search (BFS) algorithms. The execution time of these programs is competitive with a good C implementation (Section 7.2). With a few simplifications to SSSP and BFS that only affect performance, the same algorithms also run on the operators backend (Section 7.3). We also provide implementations of the remaining two algorithms, Betweenness Centrality (BC) and Connected Components (CC). While they are not as efficient as the LAGraph implementation, they demonstrate that graphalg does have the expressivity to encode them (Section 7.4).

The **key findings** of our evaluation are summarized in Section 7.5.

7.1 Experimental setup

We base our setup on the GAP benchmark suite [3].

7.1.1 System Configuration

Experiments were run on a Kubernetes pod with an allocation of 46 logical cores (Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz) and 500GB DDR3 RAM. The 46 logical cores amount to 23 CPUs, spread over 2 NUMA nodes. The graphs were stored on a 15000 RPM hard disk. No other pods were scheduled on this machine. The remaining 1 core and 12GB of RAM were reserved for Kubernetes daemon processes.

7.1.2 Software Versions

Graphalg is built on top of AvantGraph commit `93b09487`. Our implementation and benchmarking setup has the following external dependencies:

- SuiteSparse v8.0.2 [15] (the reference GraphBLAS implementation)
- LAGraph v1.0.1 [14] (only used for benchmarking)
- LLVM project (for MLIR) commit `2708869` [51]

7.1.3 Graphs

All graphs used in our evaluation are part of the GAP benchmark suite:

- road: a relatively small graph ($|V| = 24M$, $|E| = 58M$) that represents the USA road network. It has a single strongly connected component: every vertex in the graph is reachable from every other vertex. It also has an exceptionally large diameter.
- web: a crawl of all domains registered under the .sk LTD. It is the largest graph used in our evaluation ($|V| = 50.6M$, $|E| = 1,949.4M$).
- twitter: a dump of the twitter social network topology. Its size is similar to web ($|V| = 61.6M$, $|E| = 1,468.4M$), but it is noted to have a more irregular structure.
- The GAP benchmark suite also includes two very large synthetic graphs, 'kron' and 'urand'. These are too large for our current implementation to handle on this hardware, so we omit them from the evaluation.

The base data for all graphs were retrieved from the SuiteSparse Matrix Collection [16].

7.1.4 Algorithm Parameters

We follow the GAP benchmark specification and LAGraph benchmarking setup for the parameters of the algorithms we benchmark:

- Triangle Count: no additional parameters except for the graph.
- Single-Source Shortest Paths: `delta` depends on the graph. It is set to $50,000$ for road, 140 for web and 51 for twitter. We select a source node from the list included with the base graph. The same list is used in the LAGraph benchmark setup.
- PageRank: we use a damping factor of 0.85, tolerance of 10^{-4} and a maximum of 100 iterations.

- Breadth-First Search: like SSSP, We select a source node from the list included with the base graph.
- Betweenness Centrality: we select four source nodes from the list included with the base graph.
- Connected Components: no additional parameters except for the graph.

7.2 Generated Code Quality and Interpreter Overhead

In this section, we evaluate how well our optimizing compiler can transform the high-level graphalg language into efficient GraphBLAS library calls. We also measure the overhead added by the interpreter versus a fully compiled C program. For this benchmark, we take four GAP benchmark algorithm implementations from the LAGraph [35] library and port them to graphalg.

7.2.1 Algorithm Implementations

LAGraph is a collection of high-performance graph algorithm implementations built using GraphBLAS primitives. It is designed to provide high-quality implementations of common algorithms that can be used as a library, and represents the state of the art in terms of performance of GraphBLAS-based analytics. For the best possible performance, the reference GraphBLAS implementation SuiteSparse supports non-standard tuning parameters on individual matrix and vector instances to control the internal data format used for their representation. On some algorithms, particularly Single-Source Shortest Paths and Breadth-First Search, this improves performance, so the LAGraph implementation sets these hints if the implementation exposes them. These tuning parameters are advanced features, which we expect most users will not use in their algorithms. We have therefore not implemented tuning parameters in graphalg, but we do not see any difficulty in adding them for expert users. Our reference C implementations are identical to the implementations from LAGraph, except that the tuning parameters have been disabled.

The specific implementations we have benchmarked are:

- **TC**: Triangle Count using the Sandia LUT method [56]. The default implementation in LAGraph.
- **SSSP**: Delta-Stepping Single-Source Shortest Paths [50].
- **PR**: the variant of PageRank specified in the GAP benchmark specification [3], as implemented by LAGraph.
- **BFS**: Direction-Optimized Breadth-First Search [59].

The source code for the algorithms can be found in Section A.1.

7.2.2 Experimental Setup

We choose the road graph for this benchmark because it is the smallest graph. In larger graphs, the time spent inside the GraphBLAS library would take up a larger part of the running time, impeding our ability to measure the overhead of the interpreter.

An important difference with this setup compared to later ones is that we run the GraphBLAS backend in standalone mode, bypassing AvantGraph. This avoids time spent loading the graph, which is not counted in the GAP benchmarks. Fully replicating the LAGraph benchmark setup [13] also required disabling `mmap`, which AvantGraph relies on heavily.

For each benchmark instance, we loaded the graph into memory, then started with a warmup run. The instance was then run three times, of which we report the median time.

7.2.3 Analysis

See Figure 7.1 for the execution times of the algorithms on the road graph. We include the reference C implementation with and without tuning parameters. For graphalg, we include the GraphBLAS backend with and without our optimizations (as described in Section 5.2).

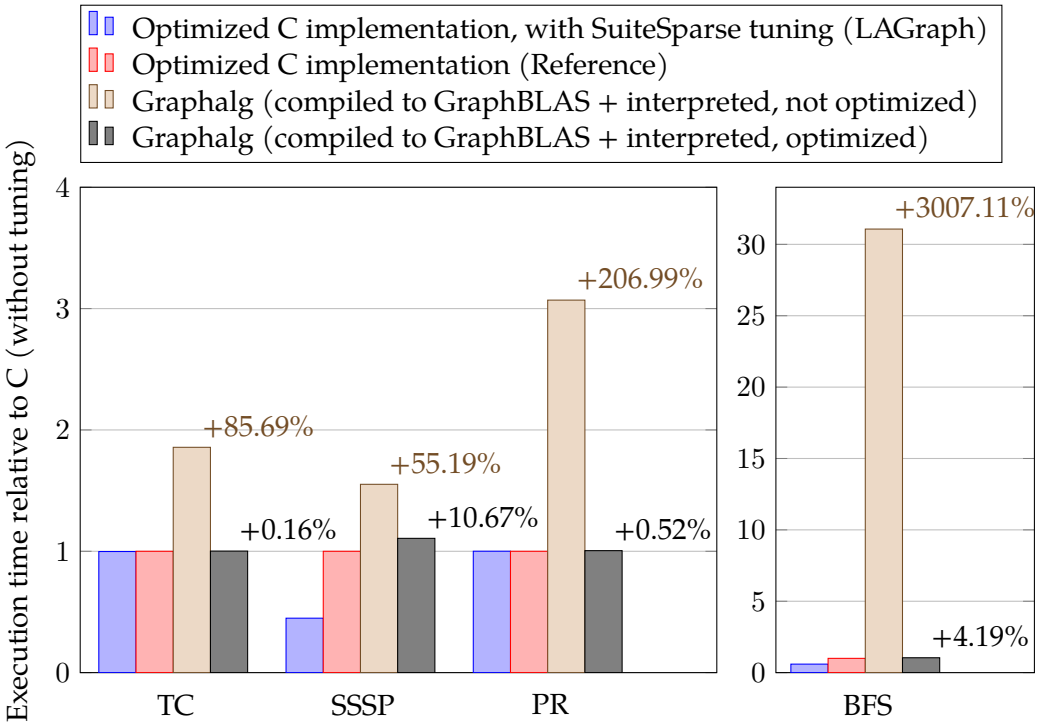


Figure 7.1: Execution time of the GraphBLAS backend compared to the reference implementation. All times are normalized to the execution time of the optimized C implementation (without SuiteSparse tuning).

We find that without optimizations, graphalg is significantly slower on all benchmarks, particularly BFS. The generated IR contains many more tensor allocations and copies, which negatively affect performance. By running GraphBLAS in debug mode[1], we can also observe that different, less efficient GraphBLAS functions are used due to a lack of fusion.

With optimizations turned on, however, execution time approaches the reference implementation and is only a few percent slower execution on most benchmarks. We also observe that the interpreter invokes the same GraphBLAS functions as the reference implementation, indicating proper fusion of operations. The difference in execution time that we measure is therefore due to the overhead of interpreting the compiled program. Overhead is highest in the SSSP program. Most likely this is because SSSP runs many short iterations with relatively cheap GraphBLAS function calls. This is in contrast to TC, where overhead is low because the program compiles into a handful of GraphBLAS calls without any control flow. PR represents a middle ground with a few tens of iterations that are moderately expensive, and hence the measured overhead is between TC and SSSP.

[1]We enable the *burble*, which causes the specific implementations used for each function to be printed to the console.

7.2.4 Limitations of the GraphBLAS backend

For the programs we have tested, the graphalg compiler produces comparable code to the optimized C implementation. The optimizations we have implemented are generic and not specific to any algorithm, but they were developed to improve the performance of these specific algorithms. In Subsection 7.4.1 we show that our optimizations also suffice for the BC and CC algorithms. We developed these algorithms after the optimizations had already been implemented, suggesting that the same optimizations would work for many other algorithms. However, we have not investigated the optimization space in detail, and additional optimizations may be needed for new algorithms.

Our current GraphBLAS backend does not make use of SuiteSparse tuning parameters. This can have a significant performance impact, as shown in Figure 7.1. Future work may investigate if the GraphBLAS backend could automatically perform such tuning, either through IR analysis or profile-guided optimization. Another solution would be to extend the language with pragmas, so expert users can set these tuning parameters manually.

7.3 GraphBLAS vs. Operators Backend

This section compares the performance of the GraphBLAS and operators backends based on total execution time, loading time and peak memory consumption.

7.3.1 Algorithm Implementations

We use the same algorithms as in Section 7.2. The implementations of SSSP and BFS have been simplified due to limitations of the operators backend, most notably the lack of conditional branching. Particularly on the fully connected road graph, this significantly impacts the performance of the SSSP program. To avoid excessive running times, we limit the implementation to 200 hops, which is still sufficient to obtain the shortest paths on all other graphs. The updated implementations can be found in Section A.2.

7.3.2 Experimental Setup

To import the graphs in AvantGraph, we converted each Matrix Market file into a list of sorted triplets (by row and column index). These triplets were then imported using the standard AvantGraph loader. Edge weights were attached as integer properties.

Contrary to the previous benchmark, the graph is read from disk every run. When measuring execution time, we include the time to read the graph from disk as well as the time needed to transform it into a representation suitable for graph analytics. Like most databases, AvantGraph is disk-based and can support graphs that are larger than main memory. Data is not assumed to be resident in main memory, so the cost of loading it is an important factor in measuring query performance.

We do not count the time needed to compile algorithms and plan the execution. In our benchmarks, this is a negligible amount, so we only measure the execution time of the query, starting from the moment that it is handed off to the runtime.

We test three different backend configurations:

- GraphBLAS backend: the same implementation as tested in Section 7.2, now integrated into AvantGraph.
- Operators backend: uses the operators backend, with semiring operations converted into a chain of join, apply and aggregate operators. Due to the lack of an anti-join operator, BFS cannot be run in this configuration.

- Operators backend, with semiring operator: uses the same operators backend, but converts semiring operations into a specialized semiring operator. This configuration does support running BFS, but because a Matrix-Vector product is not implemented, it falls back to a join-apply-aggregate chain for PR. We do not present results on PR for this configuration, since they are the same as the regular operators backend configuration.

For the benchmarks of execution and loading, we run each benchmark instance three times and report the median. Peak memory consumption does not vary significantly between different runs, so we only run each instance once.

7.3.3 Total Execution Time

In Figure 7.2 we show the query execution time on the road graph, normalized to the execution time of the GraphBLAS backend.

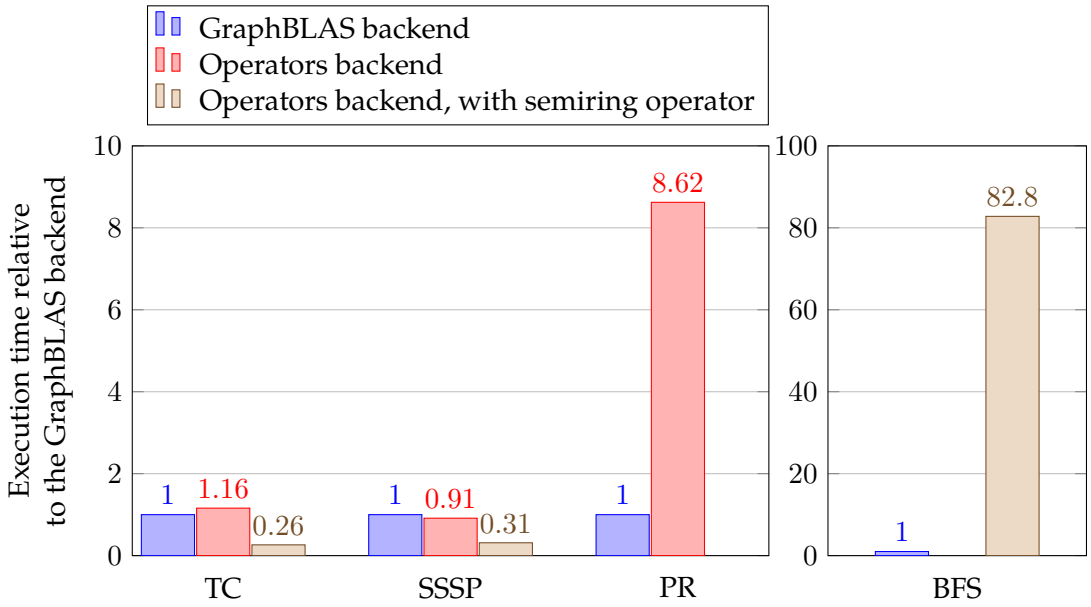


Figure 7.2: Query execution time on the road graph, relative to the GraphBLAS backend.

We draw the following conclusions based on the results:

- There is no clear winner between the two backends. On TC and SSSP, the operators backend with semiring operator outperforms the GraphBLAS backend. As we show in Subsection 7.3.4, this is partly due to graph loading time.
- The operators backend performs much better with the semiring operator enabled. Building and probing hash tables is slow compared to a specialized sparse matrix representation. Results without the semiring operator are omitted in the larger graphs because they did not terminate within reasonable time (8 hours). Because the semiring operator does not support the Matrix-Vector multiplication needed for PR, execution time of PR is much higher on the operators backend.
- The GraphBLAS backend runs BFS more than 80 times faster than the operators backend. The large difference is due to GraphBLAS' ability to aggregate in place. In the road graph all vertices are reachable from any starting node. Over many iterations, the algorithm fills a vector the size of the number of vertices in the graph. The GraphBLAS backend creates the vector once and adds a few elements to it per iteration. The operators backend on the other hand must rebuild the vector at every iteration, resulting in unnecessary copies and aggregation.

Running the same queries on the larger web graph, shown in Figure 7.3, we see some important differences.

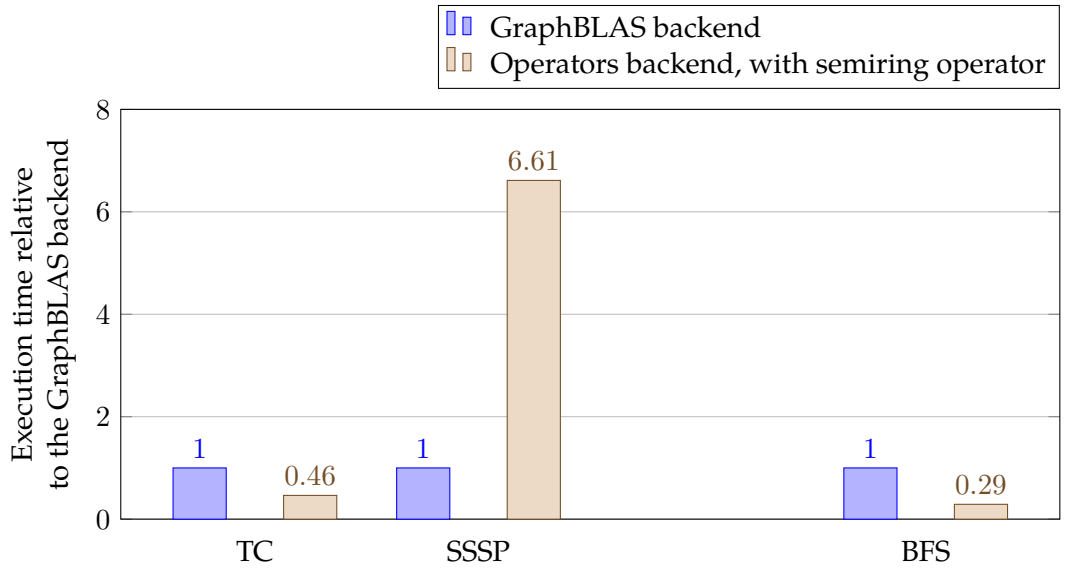


Figure 7.3: Query execution time on the web graph, relative to the GraphBLAS backend.

- SSSP now runs fastest on the GraphBLAS backend rather than the operators backend. We suspect the reason is again in-place aggregation.
- BFS is now faster on the operators backend. This is a consequence of the graph topology: the graph consists of many small connected components, so a BFS from one node will only discover some of the vertices in the graph. For the operators backend, this means there is less data to copy per iteration. The runtime of this algorithm on the web graph is dominated by the loading time, which is lower on the operators backend (see Subsection 7.3.4).

Lastly, we look at the execution times on the twitter graph, shown in Figure 7.4.

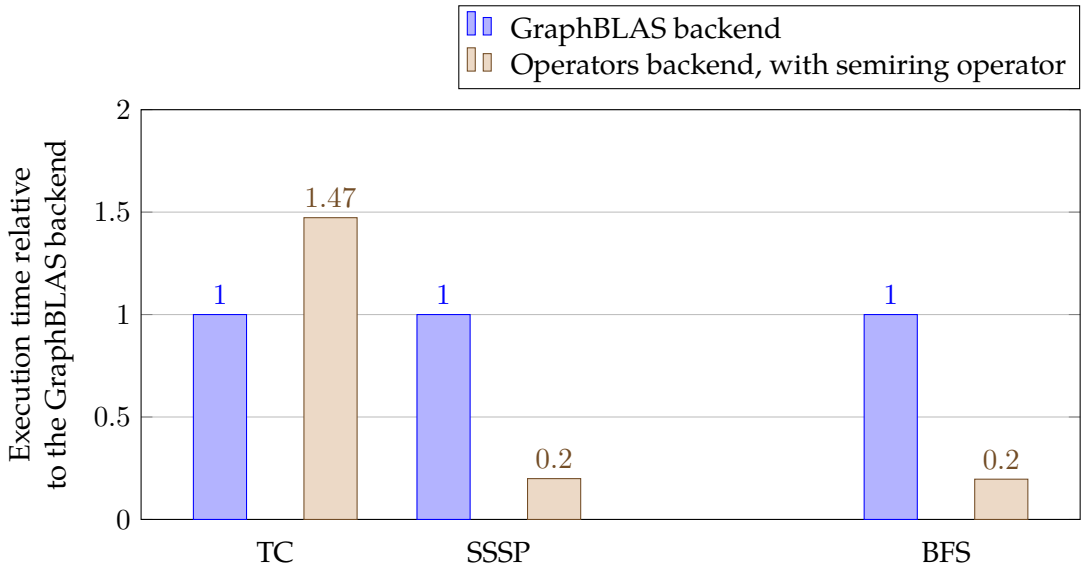


Figure 7.4: Query execution time on the twitter graph, relative to the GraphBLAS backend.

- TC is now faster on the GraphBLAS backend, where previously the operators backend consistently performed better. During execution we observed that the GraphBLAS barely utilized 2 cores. Surprisingly, the operators backend fully utilized all 46 cores, but it was still slower. Unfortunately, we cannot use profiling tools such as `perf` on the server, so we were unable to investigate this result further.
- SSSP and BFS are faster on the operators backend, again due to the loading time (see Subsection 7.3.4). Both backends spend little time executing the loop. The majority of time is spent constructing the matrix representation, which is faster on the operators backend.

In general, the operators backend performs better on short benchmarks due to quick loading. For computationally heavy benchmarks, the GraphBLAS backend performs better. In the next section, we investigate the effects of graph loading time in detail.

7.3.4 Loading Time vs. Loop Time

Now we break down the results from Subsection 7.3.3 into two parts:

- Loading time: time spent loading the graph from disk, and building an appropriate representation for executing the algorithm. In the case of the GraphBLAS backend, this involves constructing a Matrix in the internal GraphBLAS format. For the operators backend with semiring operator, we build a CSR representation directly over the input blocks, as described in Section 6.8.
- Loop time: time spent running the algorithm once the inputs have been loaded. For most algorithms, this corresponds to executing a loop. TC is an exception because it does not contain a loop. For TC, we instead count the matrix multiplication and reduction to an integer as part of the loop time. The filtering to create the L and U matrices, and building CSRs for those matrices makes up the loading time.

The results below are based on the same data as we presented earlier in Subsection 7.3.3. We compare the GraphBLAS backend to the operators backend with semiring operator. Only those benchmarks where the load time is significant are shown. We start with a breakdown of TC and SSSP on the road graph, shown in Figure 7.5.

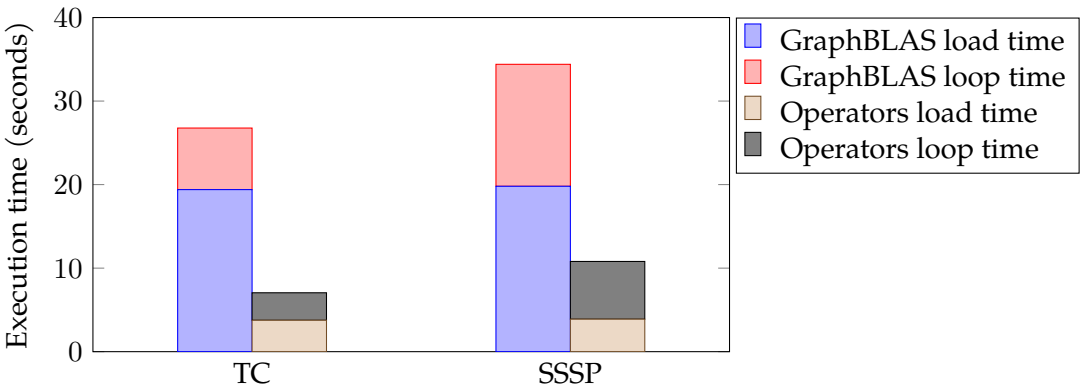


Figure 7.5: Load and loop times on the road graph.

Loading the graph takes significantly longer on the GraphBLAS backend. In both TC and SSSP, the GraphBLAS backend spends more than half of the execution time loading the graph. The operators backend finishes running the algorithm even before the algorithm starts executing on the GraphBLAS backend. Even if we disregard the loading time though, the operators backend still outperforms the GraphBLAS backend.

The results for the web graph, shown in Figure 7.6, paint a different picture.

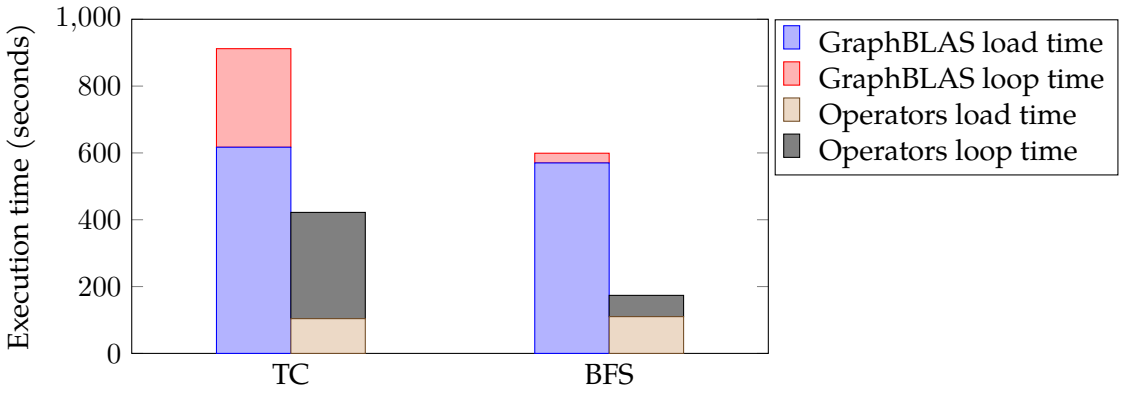


Figure 7.6: Load and loop times on the web graph.

For both TC and BFS, the GraphBLAS backend now executes the loop body faster than the operators backend. However, the loading time remains high on the GraphBLAS backend, exceeding the total execution time of the operators backend. It has already lost the race before it can start running the loop. The effect of loading time is strongest on the twitter graph, shown in Figure 7.7.

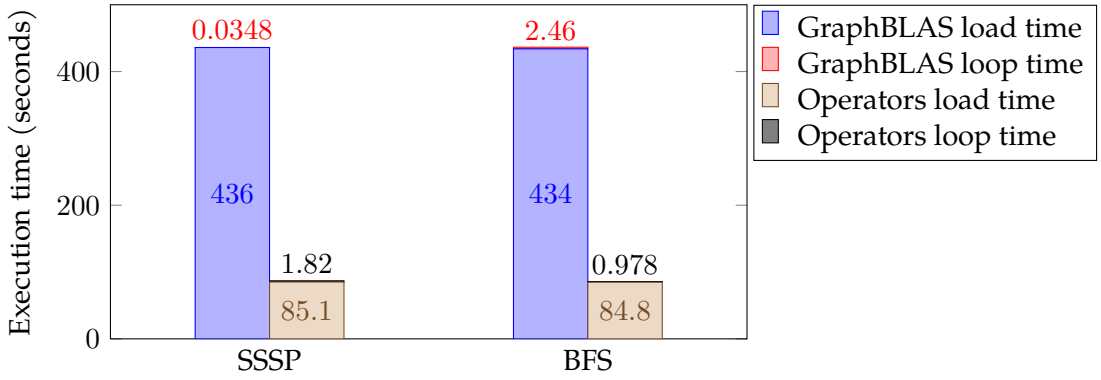


Figure 7.7: Load and loop times on the twitter graph.

In the previous two graphs TC had a low loop time, but on the twitter graph, loop time dominates the execution time. Instead, we show SSSP, where loading time does dominate execution time on this graph. Even more so than before, the loop time is negligible compared to the loading time. The size of the twitter graph is comparable to the web graph, but the effect is stronger because the two algorithms visit only a very small part of the graph.

We conclude that load time is indeed significant for half of the algorithms we have tested (although which half depends on the specifics of the graph). This is especially concerning for the GraphBLAS backend, where loading time is highest. Matrices must be converted into the GraphBLAS native format before they can be used, an unavoidable cost to using the library. On the operators backend we have more freedom to improve loading time. For example, development is underway to change the AvantGraph native storage format to be more like a CSR. This would allow the operators backend to directly run operations over the on-disk representation, eliminating loading time.

7.3.5 Memory Consumption

Lastly, we measure the peak memory consumption for each configuration. We use heaptrack [57] to log memory allocations and deallocations. Its analysis tool can then find the

point in time where memory usage peaks, and show where the memory was allocated.

We begin with the results for the road graph (3.6GB on disk), shown in Figure 7.8.

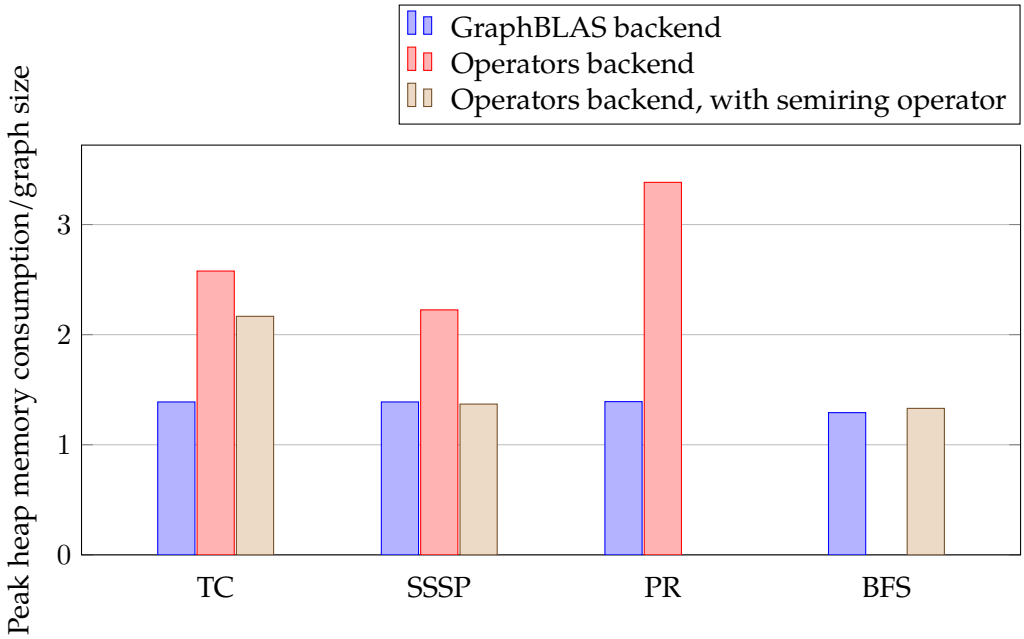


Figure 7.8: Peak memory consumption on the road graph.

We note the following about the results:

- Neither backend can operate directly on stored data. Both require the graph to fit in main memory.
- The GraphBLAS backend is remarkably efficient, consuming just a bit more than the on-disk format. Memory usage peaks during the construction of the input graph.
- Without the specialized semiring operator, the graph is represented as a hash table. This is very inefficient, consuming over twice the on-disk space. In the case of PR, multiple aggregation phases each pre-allocate a large buffer for better performance, further driving up memory usage.
- With the semiring operator enabled, the operators backend approaches the graphblas backend. TC is still less efficient because the graph is joined with itself, and the query plan constructs two CSR matrices for the graph. The GraphBLAS library appears to prevent the materialization of the intermediates.

Next, we look at the results for the web (102GB) and twitter (78GB) graphs, shown in Figure 7.9 and Figure 7.10 respectively. Because the results are so similar, the same analysis applies to both figures:

- There is no clear winner between the backends. GraphBLAS still performs better on TC because it avoids constructing two intermediate matrices, but on SSSP and BFS the operators backend is more efficient.
- On the GraphBLAS backend, memory consumption is no longer fully dominated by the graph construction phase. As a result, we observe slightly higher relative memory usage.
- The lower relative memory usage of the operators backend on SSSP and BFS is related to the design of our aggregation operator: it pre-allocates space based on the number of

vertices in the graph. The edge-to-vertex ratio is much higher on the large graphs (24-38 edges/vertex) compared to road (2.4). Because of the higher edge-to-vertex ratio, the aggregation operators use a relatively low amount of memory.

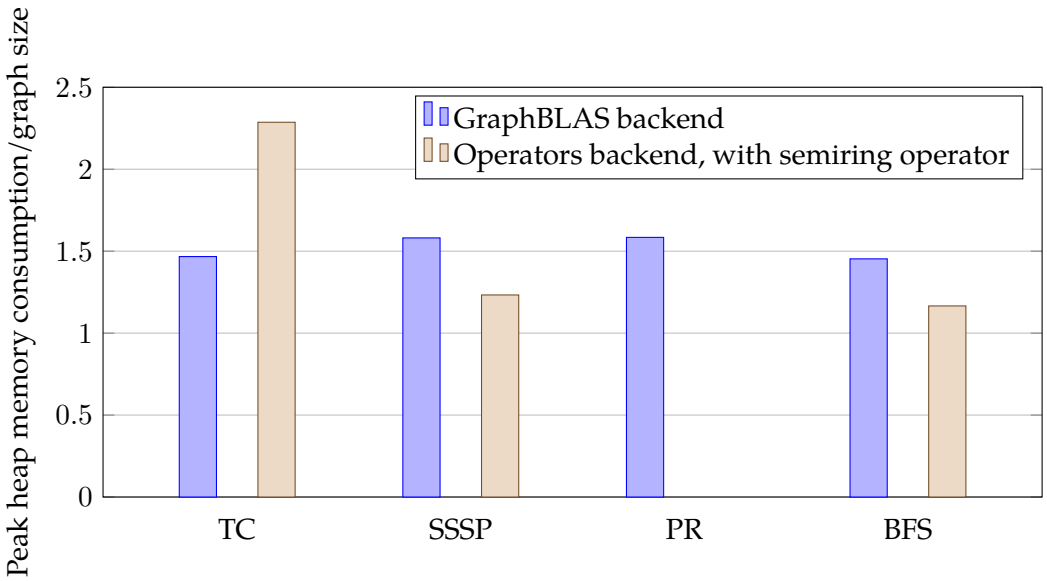


Figure 7.9: Peak memory consumption on the web graph.

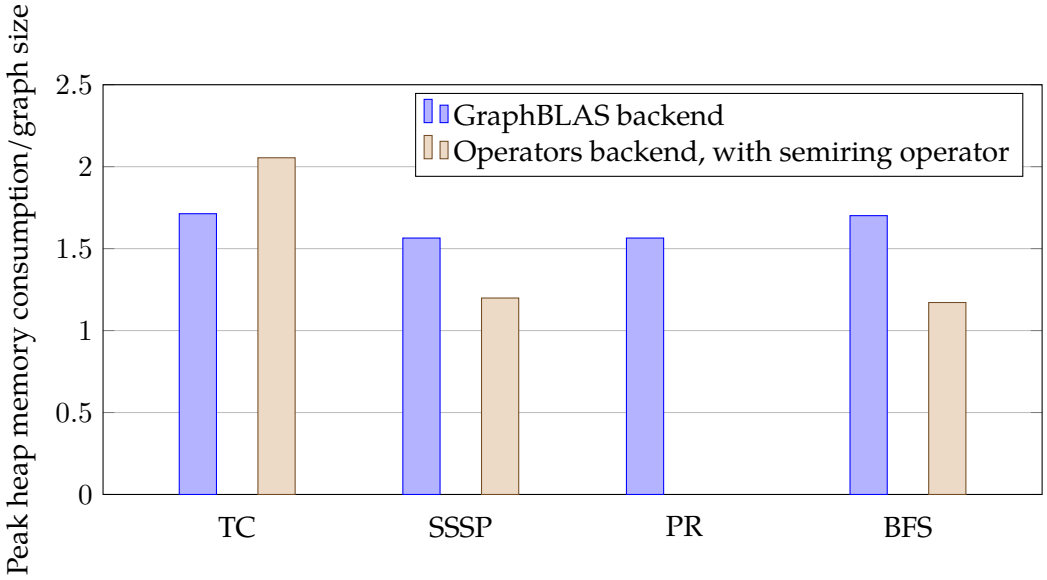


Figure 7.10: Peak memory consumption on the twitter graph.

To conclude: with a peak memory consumption on the order of the size of the input graph, the current implementations are not suitable for extremely large graphs. The high memory consumption does not come as a surprise, however. The on-disk storage format of AvantGraph is not suitable for graph analytics, so both backends are forced to load the full graph into a more suitable representation. Once the on-disk format evolves to be more amenable to analytics workloads, we expect that the operators backend will be able to directly use the on-disk format, thereby reducing memory consumption dramatically. Another issue is that our current aggregation operator pre-allocates a large buffer. The size of this buffer could be tuned better to further reduce memory consumption.

7.4 Expressing Betweenness Centrality and Connected Components

A port of the betweenness centrality (BC) and connected components (CC) algorithms is not as straightforward as it is for the other four algorithms. Although the specifics vary per algorithm, in both cases the issue arises from limitations of the GraphBLAS API, which the implementation resolves by using the flexibility of the C language.

The LAGraph implementation of BC needs to keep track of the depth at which each vertex is first seen. Instead of storing a depth level per vertex, the implementation keeps a list of bitmaps, one for each level. This approach is more efficient, but since the current version of graphalg does not support arrays, we cannot replicate this code.

For CC, LAGraph provides two implementations, but neither can be ported directly to graphalg. The fastest implementation needs a selection predicate that filters vertices based on their rank. Since this functionality is not available in the current GraphBLAS API (there is a note in the code that it will be added), it is implemented in plain C with raw access to the underlying matrix data and explicit parallelism using OpenMP. The slower implementation is also not suitable because it requires user-defined selection predicates.

While the current graphalg compiler does not support a particularly efficient implementation of BC or CC, it does have the *expressivity* to encode them. To show this, we implement slower but correct versions of both algorithms, working around the limitations of the language where necessary. For BC, we stick to the LAGraph implementation where possible. Instead of using arrays, we stack the depth matrices on top of each other in one large matrix. Per-depth matrices are extracted using matrix sub-indexing. To implement CC, we use the simple component-at-a-time algorithm.

Both algorithms are included in Section A.3, and can be executed using the GraphBLAS backend (the current operators backend does not support matrix indexing).

7.4.1 Performance Compared to LAGraph

We quantify the performance impact of using our simplified implementations by comparing their execution time to the LAGraph version, similar to Section 7.2. We also show that the existing optimizations of our compiler are sufficient to optimize our implementations of BC and CC.

Experimental Setup

Our setup closely resembles the one from Section 7.2.

- We use the smaller road graph so that the interpreter overhead is most noticeable.
- The GraphBLAS backend is used in standalone mode.
- Each benchmark instance runs with the graph already loaded into main memory.
- For each instance, we do a warmup run. We then run it three times, and report the median execution time.

We test four different configurations:

- The LAGraph implementation. This represents the best known implementation of the algorithm using GraphBLAS.
- A C implementation of our simplified algorithm. This is the algorithm as we implement it for graphalg, but written in C. We have hand-optimized the implementation by fusing operations as much as possible. It is not as fast as LAGraph because we do not use

native C features like arrays or raw matrix access. Our implementation also does not use SuiteSparse tuning parameters.

- The graphalg version of the algorithm, executed by the GraphBLAS backend with optimizations disabled. We show this configuration to demonstrate the effect of our optimizations.
- The graphalg version of the algorithm, executed by the GraphBLAS backend with optimizations enabled. This is the best known implementation of the algorithm using the current version of graphalg.

Analysis

Figure 7.11 shows the performance of the different configurations.

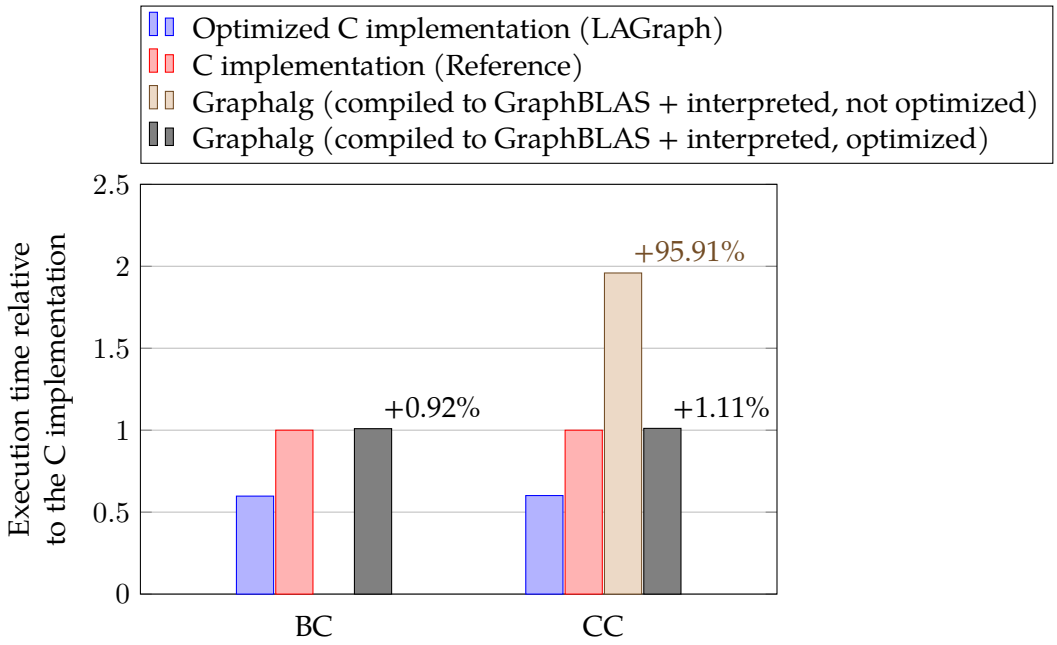


Figure 7.11: Execution time of BC and CC on the GraphBLAS backend compared to LAGraph. All times are normalized to the execution time of the C implementation. Results for BC on Graphalg without optimizations are omitted because the interpreter ran out of memory.

Based on these results, we conclude the following:

- The LAGraph implementations are indeed significantly faster than our simplified algorithm. We expect that the difference would be even larger on bigger graphs. For BC, this would involve larger copies of submatrices. For CC, more repeated graph traversals would be necessary.
- Without optimizations, graphalg is much less efficient than the C implementation. CC takes twice as long to execute. For BC, the interpreter even runs out of memory after allocating more than 500 GB.
- With optimizations turned on, performance is comparable to the C implementation. This shows that our optimizations also perform well on programs that they have not been specifically designed for. We observe that the GraphBLAS backend issues the same GraphBLAS calls as the C implementation. The only difference in execution time comes from interpreter overhead.

We conclude that both BC and CC can be expressed in the current version of graphalg, although performance could be improved further. An efficient implementation of BC and CC would be possible with reasonable extensions to the language. To support BC, arrays or three-dimensional tensors would be a suitable addition. For CC, rank-based selection predicates would suffice. We leave these additions as future work.

7.5 Key Findings

Our key findings from the evaluation are the following:

- The majority of algorithms from the GAP benchmark suite can be straightforwardly ported from LAGraph to graphalg. Barring the advanced tuning settings available in specific GraphBLAS implementations, our GraphBLAS backend is competitive with a state-of-the-art implementation in C (Section 7.2). The optimizations we have developed are also effective on algorithms that they were not specifically designed for (Subsection 7.4.1).
- The operators backend performs much better with the semiring operator enabled. The CSR representation for matrices is faster to build and probe than a hash table (Subsection 7.3.3). It also uses less memory (Subsection 7.3.5).
- Between the GraphBLAS and operators (with semiring operator) backends, there is no clear winner. The operators backend is faster on short benchmarks. On more computationally intensive problems, the GraphBLAS backend shows superior performance (Subsection 7.3.3).
- The operators backend performs poorly on algorithms that fill a large vector over many iterations, due to a lack of in-place aggregation (Subsection 7.3.3). Given the severe performance impact, future work is needed.
- On a disk-based system such as AvantGraph, the time needed to load the graph from disk and represent it in a way suitable for analytics, *load time*, is an important factor in query performance. For half of the algorithms tested, we find that load time makes up a significant part of the total execution time (Subsection 7.3.4). It is particularly high for the GraphBLAS backend, and non-trivial to reduce. On the operators backend load time is consistently lower. Upcoming changes to the on-disk representation of graphs are expected to further reduce the operators backend load time.
- Load time is significant in both backends because the entire graph is loaded into main memory. To allow querying graphs larger than main memory, future work is needed. In particular, the representation of graphs on disk must be changed to support direct use in graph algorithms. Work is ongoing to make these changes in AvantGraph.
- Graphalg is expressive enough for all algorithms included in the GAP benchmark suite. For four out of six algorithms, we can implement state-of-the-art versions with excellent performance (Section 7.2). The other two algorithms can be expressed, albeit not very efficiently (Section 7.4). We posit that these shortcomings can be addressed with reasonable and generally useful extensions to the language.

Chapter 8

Related Work

In this chapter, we discuss research and systems that are related to our work along one or more of the following axes:

- User-provided algorithm support in databases.
- Domain-specific languages for graph algorithms.
- Linear algebra in relational databases.
- MLIR-based software related to linear algebra or databases.
- User-friendly APIs built on top of GraphBLAS.

8.1 Algorithm Support in Databases

We begin our treatment of algorithm support in databases by noting that many database systems have supported procedural extensions to SQL for decades. This technically makes it possible to implement algorithm in them, but as noted by Gupta and Ramachandra [22], these languages are rarely efficient. Therefore, in the remainder of this section, we focus on systems that go beyond such extensions.

A prominent example among graph databases is the commercial system TigerGraph [17]. TigerGraph's query language GSQL is based on SQL, with extensions to support accumulators and imperative control flow. The extensions make GSQL expressive enough to encode implementations for the algorithms in the GAP benchmark suite [54]. We estimate that the expressivity of GSQL is similar to that of graphalg. One limitation of GSQL, which graphalg does not have, is that per-vertex accumulators must be defined in the global scope, and may not be declared inside loops. Being a commercial system, implementation details of TigerGraph's planner and runtime are not publicly available, so we can only speculate how much their system resembles graphalg internally. Publicly available documentation makes no mention of any connections to linear algebra. The paper introducing GSQL [17] mentions Green-marl [23] as a source of inspiration, and GSQL indeed appears to resemble a vertex and edge-set-based language, given that accumulators originate from this model. We like that TigerGraph shows that in-place accumulators can be added to a database engine. We hope to implement similar functionality in our operators backend in future work.

Neo4J is another graph database with some support for user-defined code [53]. As briefly mentioned in Section 3.2, users of the database can write custom procedures in Java. The Cypher language includes a dedicated `CALL` keyword to invoke such procedures. The Java API offers programmers great flexibility and expressiveness, but it also makes it impossible for the database to optimize the implementation. In Neo4J, procedures always access the graph directly. They can not consume inputs from subqueries like in graphalg.

User-defined operators [48] (UDO) have been proposed to address this very problem, albeit in a relational database setting. The UDO framework lets users implement custom operators that can be embedded at any position in the query. Inputs and outputs are produced just like the native operators provided by the database. Custom operators, implemented in C++, are compiled together with the rest of query, which allows for embedding with virtually zero overhead. The UDO solution is appealing because it offers both flexibility and very high performance. There is also no need to implement a new language with an optimizing compiler, as we have done for graphalg. An important downside of the UDO framework is that since operators are written in C++ and are not sandboxed, a bug in a custom operator can crash the database. It can also corrupt intermediate results of concurrent queries or internal data structures used by the database. Another downside is that the query planner does not know the behavior of a custom operator, so it can not apply any optimizations to it. In this regard, it is similar to our GraphBLAS backend.

8.2 Domain-specific Languages for Graph Algorithms

Oracle PGX [7], formerly Greenmarl [23], is a DSL for graph analytics. The language is based on the vertex and edge set model. Like TigerGraph it has explicit accumulators, but it is not as deeply integrated with the database. While PGX executes its algorithms on the same runtime as the query language PGQL [43], both languages have separated compiler pipelines. Algorithms and queries can thus be evaluated on the same dataset, but algorithms cannot be embedded in queries like graphalg. PGX is most interesting to us because the design goals are similar. PGX strives to be a language that is efficient yet easy to use, and safe to run inside a database.

For similar reasons, the GraphIt [60] language is interesting to us. It has demonstrated excellent performance on many graph algorithms, also compared to Greenmarl. The language, based on the vertex and edge set model, stands out by separating edge processing logic from graph traversal. This allows the compiler to optimize the traversal strategy based on graph properties. For example, the GraphIt compiler can apply the push-pull optimization automatically, which must be done manually in GraphBLAS [59]. We suspect there is a significant overlap between the optimizations GraphIt performs and those employed by a query planner. Future work may investigate to what extent such optimizations could be automatically applied to graphalg.

8.3 Linear Algebra in Graph Databases

RedisGraph [9] is a graph database provided as a Redis module. It is of particular interest to us because the database is built on top of the GraphBLAS library: graphs are stored as matrices in the native GraphBLAS format, and queries are executed using standard GraphBLAS library calls. RedisGraph exposes only the Cypher interface to its users, so running algorithms on it is not possible. However, the GraphBLAS backend could be easily adapted to integrate with RedisGraph. Because the graph is already stored in the native GraphBLAS format, it would fully eliminate the graph loading times observed in the AvantGraph integration. Unfortunately, RedisGraph is being phased out [27], so it is unlikely such an integration will be attempted.

While we have mostly discussed implementations so far, there is also relevant research on the theoretical expressiveness of linear algebra for use in graph databases. MATLANG [8] is a formal language with linear algebra primitives and other common matrix operations. The authors show that the basic version of MATLANG is equivalent to relational algebra extended with arithmetic and aggregation. During the implementation of graphalg, whose expressiveness subsumes that of basic MATLANG, we have indeed found that arithmetic

and aggregation had to be added to AvantGraph to support it. While the extensions to MATLANG presented in the original paper have no analog in our language, there is other work extending MATLANG to include loops [19], bringing MATLANG closer to graphalg in terms of expressive power. We hope that based on the work done on MATLANG, a more formal definition of the expressiveness of graphalg can be established in the future.

8.4 Linear Algebra in Relational Databases

Linear algebra extensions to relational databases have been widely studied [45]. Given the volume of related work on this topic, we restrict our review to systems designed for sparse rather than dense matrices.

The first implementation of sparse matrices in a relational database system that we are aware of is Sparse Relational Array Mapping [10] (SRAM). SRAM is integrated into MonetDB/X100 [6]. SRAM shows many similarities with our Operators backend. For example, sparse matrices are also represented as blocks with index columns and a value column. A key difference is that SRAM does not provide linear algebra operations in the query language. Instead, SRAM exposes more basic array manipulation operations, which can be combined to implement e.g. matrix multiplication. The mapping to relational algebra is slightly more complicated in SRAM because it adopts different semantics for entries in matrices without an explicit value. In graphalg, such entries are simply omitted from the tuple stream, but in SRAM they have an implicit default value. If used in a join, entries with the default value may satisfy the join condition and thus need to be part of the output. As a result, the element-wise product of two matrices requires only one join in graphalg, whereas in SRAM it requires a union of 4 joins. SRAM supports persisting arrays to disk and indexing them, which AvantGraph/graphalg does not support yet.

Another more recent proposal to introduce linear algebra concepts in relation databases is presented in [31]. The interface is quite different from SRAM (and graphalg). Instead of placing matrices at the level of tables, new column types are introduced for matrices and vector, so that matrices are stored inside tables. Additional functions to operate over the new types, such as matrix multiplication, are provided. There are also conversion functions to convert a table into a matrix, and the authors note that the system can also support hybrid formats. For example, a matrix may be stored as a table of vectors if this makes the query easier to express or more efficient to execute. Contrary to SRAM graphalg, vectors are always stored as dense arrays. Matrices are stored as a list of vectors, with consecutive empty rows compressed using run-length encoding. Because matrices are stored as values in tables, linear algebra operations are performed inside of projections, just like scalar arithmetic. Consequently, the query optimizer does not perform any optimization on such operations. The projection operator executes linear algebra expressions exactly as they were written, using a dense linear algebra library.

8.5 MLIR for Linear Algebra and Databases

In this section, we present projects with a connection to our work that also use the MLIR framework.

While originally designed for dense matrices, the MLIR tensor type has been extended with support for sparse representations [4]. With these changes, it is possible to attach an attribute to tensor types indicating that they are sparse. Because graphalg does not mix dense and sparse tensor types, this new annotation is not relevant to our implementation. What is relevant however is that the built-in linear algebra operations that come with MLIR have been updated to support processing sparse matrices. Thanks to this effort, there is a clear path towards compilation of graphalg programs to native code, with significantly reduced

implementation effort. For the GraphBLAS backend, we still choose to use an interpreter, as fully compiled programs are more difficult to debug, and execution time is dominated by GraphBLAS library calls. Because AvantGraph does not compile its queries to native code, it is also not a good fit for the operators backend.

Moving from MLIR infrastructure to tools built using MLIR, COMET [55] is a compiler for sparse tensor algebra based on MLIR. With an extension to support semirings [21], it is also useful for graph analytics. In a head-to-head comparison with LAGraph, COMET demonstrates that full compilation offers significantly higher performance. COMET is also interesting because it includes a domain-specific language to write algorithms in. It operates at a similar abstraction level to graphalg, and before deciding to implement graphalg, we considered using the COMET DSL instead. Unfortunately, it lacks support for control flow. As mentioned in [4], COMET predates sparse tensor support in MLIR. It uses custom internal dialects to handle sparse tensors rather than the new features in MLIR, which we fear may hurt community adoption. If a fully compiled version of graphalg is ever built, we think COMET will be a fantastic source of inspiration.

LingoDB [24] is a data processing system built on MLIR. LingoDB supports fully compiled execution of SQL queries, but unlike other such systems, it is designed to be maintainable and extensible. A SQL query is converted into MLIR directly after parsing and undergoes multiple lowerings towards LLVM IR. It is then JIT-compiled and executed to produce the query results. While AvantGraph is not a fully-compiled engine, and we mostly use MLIR to represent IR which is not yet relational, we have found LingoDB a very useful source of inspiration when developing the IPR dialect.

8.6 User-Friendly Interfaces for GraphBLAS

The GraphBLAS standard defines a C API for linear algebra operations. While powerful, the API is not particularly ergonomic to use, owing to the limitations of the C language. To create a more convenient interface to GraphBLAS, bindings to the high-level languages Python and Julia have been developed [42]. Experiments done by the authors show that writing the algorithm in a high-level language does not have a significant impact on the performance, because the computationally expensive operations are handled by GraphBLAS. This is consistent with our evaluation of the GraphBLAS backend.

Chapter 9

Discussion

Before concluding this thesis, we discuss the limitations of our work, and how they may be addressed in future work. Most of the limitations we identify are contingent only on engineering effort, while some also require additional research.

9.1 Loading Time and Memory Consumption

Our current system has significant runtime overhead and high memory consumption because the full graph must be loaded into the main memory before it can be used in an algorithm. The underlying cause is that the current on-disk graph storage format used by AvantGraph is not well suited to graph analytics. AvantGraph currently stores edges as an unordered list of source and target pairs. Most graph algorithms need fast access to the neighbors of a vertex, which is not possible with the current on-disk representation. Both backends are thus forced to read the entire graph and build a more efficient representation.

This problem is especially difficult to solve in the GraphBLAS backend. The GraphBLAS library is designed for in-memory analytics and has no API for performing operations on data that is not stored in its internal formats. It may be possible to extend it with support for a disk-backed format, but since the implementations of the operations are tightly linked to the representation, we expect it would require major changes throughout the code base.

For the operators backend, we are more optimistic. The only changes needed here to operate over a disk-backed graph would be in the matrix probe operator. Work is currently ongoing to migrate the on-disk storage into a format that resembles CSR. After this migration, the matrix probe operator can be changed to read directly from disk rather than an in-memory CSR, virtually eliminating loading time.

9.2 In-place Aggregation in the Operators Backend

A common pattern in graph algorithms is to iteratively populate or aggregate into a vector. The GraphBLAS backend can perform such operations efficiently in place. On the operators backend, however, all existing entries must be fed into a union operator with the new values, which is unnecessarily costly. Some systems, like TigerGraph [17], support explicit accumulators to solve this problem. In future work, we hope to investigate automatically inserting accumulators as a compiler optimization.

9.3 Missing Functionality in the Operators Backend

The operators backend does not currently have feature parity with the GraphBLAS backend. Because graphalg is inspired by GraphBLAS, many operations have a direct mapping to the

GraphBLAS API. On the operators backend, however, we have to do significantly more work to transform the program into equivalent IPR. In some cases, we have also had to implement additional functionality in IPR and the runtime. As a result, adding features to the operators backend takes more effort. We do believe the effort is worthwhile though, since the benefit is better integration with the database. We expect that future work will focus mainly on the operators backend, and this feature parity will be achieved.

9.3.1 Conditional branching

While the operators backend supports loops with an early break condition, we have not implemented support for conditional branching (if statements). All programs from the GAP benchmark suite can be expressed without conditional branching, but we have had to simplify some programs to eliminate branching, leading to less efficient implementations of those algorithms. Branching is non-trivial to implement in the AvantGraph runtime because of the push-based evaluation model (see Section 2.3). In the push-based model, operators do not wait until they are requested to produce outputs, but rather they proactively produce results and push them to the operators that depend on them. A naive implementation of an 'if' operator is shown in Figure 9.1.

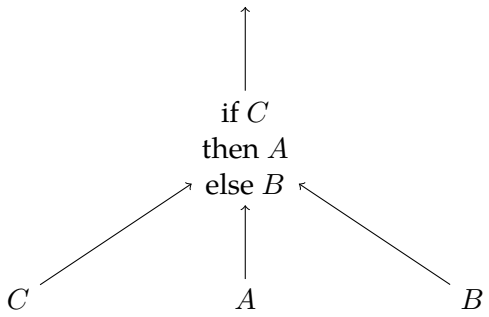


Figure 9.1: Naive execution plan with a conditional branch.

The desired semantics of the if operator is that C is evaluated first, after which either A or B is evaluated, but not both. In a push-based model however, C, A and B will all be evaluated concurrently, resulting in speculative execution of both branches. While correct behaviour, speculative execution can waste resources. AvantGraph also does not have existing support for canceling the execution of operators, which we would need to avoid completing a branch whose results are no longer needed. These problems are not fundamental, but they did make the cost of implementing an if operator too high for this project.

9.3.2 Multiple Return Values from Loops

IPR is designed to represent trees of operators that each return a single stream of tuples. This conflicts with for loops in graphalg, which may have multiple result values. Initially we attempted to add support for multiple return values from expressions specifically for this use case. However, we have found that the consumers of IPR rely on the single result property, particularly the query optimizer, so we have had to abandon this approach. To maintain the tree structure in the presence of loops, we have found it necessary to duplicate loops, creating a copy for each result value. In the programs we have tested, loops rarely have more than a single result value, so this approach is sufficient. We hypothesize that it would be possible to fuse them again in the execution plan generation step after all optimization of IPR has been performed. Implementing this is left as future work.

9.3.3 No left/anti-join

AvantGraph did not support the execution of anti-joins when we started this project. Without an anti-join, it is impossible to express complemented masks, a necessary primitive to implement e.g. Breadth-First Search. We estimated that implementing a fully generic anti-join for tuple streams with an arbitrary amount of columns and data types would require significant effort. Therefore, we implemented limited support into the custom semiring operator. This is enough for our benchmark programs, but more work is required to support complemented masks independent of semiring operations.

9.3.4 Unsorted base table data

As mentioned in Section 6.8, our current solution requires that the edges of the graph are stored on disk in sorted order. Other components of AvantGraph do not rely on this guarantee, and the AvantGraph data importer makes no effort to maintain it. Our current solution is to manually sort the input graph before inserting it. Since the AvantGraph team is already working on a new on-disk format that will guarantee sorted order, we feel that this approach is sufficient until the new format is ready.

9.4 Efficiently Expressing More Algorithms

Our benchmark setup includes only six algorithms, four of which we can currently express efficiently. As we have established in Section 7.4, this is not a fundamental limitation of graphalg, but rather it is a matter of careful language design and compiler engineering. Besides betweenness centrality and connected components, in future work, we would like to investigate community detection using the Louvain [5] and Infomap [44] methods[1].

9.5 Comparison with Other Graph Databases

We would have liked to compare our system to other graph databases with support for user-defined algorithms. Unfortunately, the only systems that support this, TigerGraph [17] and Oracle PGX [7], are both commercial and not freely available for evaluation. Even among relational databases, there are no obvious candidates to compare to. We believe Umbra with its user-defined operators [48] would be the best fit, but it is not publicly available either. We hope that in the context of a conference or journal paper, it will be possible to obtain one of these systems for evaluation.

In summary, most of the current limitations of our compiler can be addressed with additional engineering effort. For in-place aggregation and returning multiple results from loops, we expect that additional research is needed as well. In the next chapter, we conclude the thesis.

[1]The Louvain and Infomap methods were highlighted as important algorithms by an industry partner.

Chapter 10

Conclusion

Our goal for this project was to increase the programmability of graph databases by adding support for user-defined graph algorithms that can be embedded into queries.

We have developed graphalg, a domain-specific language for writing graph algorithms in the language of linear algebra. Our graphalg compiler, integrated into the AvantGraph graph database, includes two backends. The first is the GraphBLAS backend, which compiles graphalg into GraphBLAS library calls and executes them in an interpreter. The operators backend adopts a radically different approach, instead compiling graphalg into the internal IR of the database, where it is optimized and executed as one with the query.

Graphalg is a high-level language with automatic memory management and parallelization. Based on linear algebra, we believe it is easy to learn and use. In exchange for this ease of programming, graphalg sacrifices little in terms of performance. We have shown that it is nearly as efficient as an optimized and fully compiled C implementation of the same algorithm, thanks to the compiler optimizations we have implemented. Graphalg is also a safe language to embed in a database, thanks to its memory safety and guaranteed termination. The language is expressive enough for a variety of applications, which we have shown by implementing graphalg versions of all algorithms included in the GAP benchmark suite.

Today, graph analytics is usually performed with specialized tools. Even if the data is originally stored in a graph database, it is exported and processed in an external tool. As our work demonstrates, existing graph databases can be adapted to support the efficient execution of analytics workloads. Using our system, a graph database can offer the same functionality as an external graph analytics framework with a more convenient interface and excellent performance.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. eng. Addison-Wesley series in computer science. OCLC: 12285707. Reading, Mass.: Addison-Wesley Pub. Co., 1986. Chap. 2.1. ISBN: 978-0-201-10088-4. URL: http://www.gbv.de/dms/bowker/toc/9780201100884.pdf (visited on 05/15/2023).

[2] Ariful Azad et al. "Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite". In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2020, pp. 216–227. DOI: 10.1109/IISWC50251.2020.00029.

[3] Scott Beamer, Krste Asanović, and David Patterson. *The GAP Benchmark Suite*. arXiv:1508.03619 [cs]. May 2017. DOI: 10.48550/arXiv.1508.03619. URL: http://arxiv.org/abs/1508.03619 (visited on 11/24/2022).

[4] Aart Bik et al. "Compiler Support for Sparse Tensor Computations in MLIR". In: *ACM Transactions on Architecture and Code Optimization* 19.4 (Sept. 2022), 50:1–50:25. ISSN: 1544-3566. DOI: 10.1145/3544559. URL: https://dl.acm.org/doi/10.1145/3544559 (visited on 07/22/2023).

[5] Vincent D. Blondel et al. "Fast unfolding of communities in large networks". en. In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/P10008. URL: https://dx.doi.org/10.1088/1742-5468/2008/10/P10008 (visited on 07/26/2023).

[6] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 225–237. URL: http://cidrdb.org/cidr2005/papers/P19.pdf (visited on 07/22/2023).

[7] Houda Boukham et al. "Spoofax at Oracle: Domain-Specific Language Engineering for Large-Scale Graph Analytics". In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASIcs). ISSN: 2190-6807. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 5:1–5:8. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASIcs.EVCS.2023.5. URL: https://drops.dagstuhl.de/opus/volltexte/2023/17775 (visited on 04/11/2023).

[8] Robert Brijder et al. "On the Expressive Power of Query Languages for Matrices". In: *ACM Transactions on Database Systems* 44.4 (Oct. 2019), 15:1–15:31. ISSN: 0362-5915. DOI: 10.1145/3331445. URL: https://doi.org/10.1145/3331445 (visited on 07/22/2023).

[9] Pieter Cailliau et al. "RedisGraph GraphBLAS Enabled Graph Database". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 285–286. DOI: 10.1109/IPDPSW.2019.00054.

[10] Roberto Cornacchia et al. "Flexible and efficient IR using array databases". en. In: *The VLDB Journal* 17.1 (Jan. 2008), pp. 151–168. ISSN: 0949-877X. DOI: 10.1007/s00778-007-0071-0. URL: https://doi.org/10.1007/s00778-007-0071-0 (visited on 07/22/2023).

[11] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://dl.acm.org/doi/10.1145/115372.115320 (visited on 05/15/2023).

[12] Timothy A. Davis. "Algorithm 10xx: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Trans. Math. Softw.* (2022).

[13] Timothy A. Davis. *LAGraph (src/benchmark)*. original-date: 2019-01-29T16:39:28Z. July 2023. URL: https://github.com/GraphBLAS/LAGraph (visited on 07/25/2023).

[14] Timothy A. Davis. *Release v1.0.1 · GraphBLAS/LAGraph*. en. URL: https://github.com/GraphBLAS/LAGraph/releases/tag/v1.0.1 (visited on 08/03/2023).

[15] Timothy A. Davis. *Release v8.0.2 (June 29, 2023) · DrTimothyAldenDavis/GraphBLAS*. en. URL: https://github.com/DrTimothyAldenDavis/GraphBLAS/releases/tag/v8.0.2 (visited on 08/03/2023).

[16] Timothy A. Davis and Yifan Hu. "The university of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://dl.acm.org/doi/10.1145/2049662.2049663 (visited on 07/25/2023).

[17] Alin Deutsch et al. "Aggregation Support for Modern Graph Analytics in TigerGraph". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, May 2020, pp. 377–392. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3386144. URL: https://dl.acm.org/doi/10.1145/3318464.3386144 (visited on 07/22/2023).

[18] Peter Eisentraut. *SQL:2023 is finished: Here is what's new*. en. Apr. 2023. URL: http://peter.eisentraut.org/blog/2023/04/04/sql-2023-is-finished-here-is-whats-new (visited on 04/26/2023).

[19] Floris Geerts et al. "Expressive Power of Linear Algebra Query Languages". In: *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS'21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 342–354. ISBN: 978-1-4503-8381-3. DOI: 10.1145/3452021.3458314. URL: https://dl.acm.org/doi/10.1145/3452021.3458314 (visited on 07/22/2023).

[20] G. Graefe. "Volcano/spl minus/an extensible and parallel query evaluation system". In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135. DOI: 10.1109/69.273032.

[21] Luanzheng Guo et al. "Towards Supporting Semiring in MLIR-Based COMET Compiler". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 542–543. ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569683. URL: https://dl.acm.org/doi/10.1145/3559009.3569683 (visited on 07/22/2023).

[22] Surabhi Gupta and Karthik Ramachandra. "Procedural extensions of SQL: understanding their usage in the wild". In: *Proceedings of the VLDB Endowment* 14.8 (Apr. 2021), pp. 1378–1391. ISSN: 2150-8097. DOI: 10.14778/3457390.3457402. URL: https://dl.acm.org/doi/10.14778/3457390.3457402 (visited on 07/22/2023).

[23] Sungpack Hong et al. "Green-Marl: a DSL for easy and efficient graph analysis". In: *ACM SIGPLAN Notices* 47.4 (Mar. 2012), pp. 349–362. ISSN: 0362-1340. DOI: 10.1145/2248487.2151013. URL: https://dl.acm.org/doi/10.1145/2248487.2151013 (visited on 07/11/2023).

[24] Michael Jungmair, André Kohn, and Jana Giceva. "Designing an Open Framework for Query Optimization and Compilation". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2389–2401. URL: https://www.vldb.org/pvldb/vol15/p2389-jungmair.pdf (visited on 05/16/2023).

[25] Jeremy V. Kepner and J. R. Gilbert. *Graph algorithms in the language of linear algebra*. en. Software, environments, and tools. Philadelphia: Society for Industrial and Applied Mathematics, 2011. ISBN: 978-0-89871-990-1.

[26] Arijit Khan. *Vertex-Centric Graph Processing: The Good, the Bad, and the Ugly*. arXiv:1612.07404 [cs]. Dec. 2016. DOI: 10.48550/arXiv.1612.07404. URL: http://arxiv.org/abs/1612.07404 (visited on 07/11/2023).

[27] Lior Kogan and Pieter Cailliau. *RedisGraph End-of-Life Announcement*. en. July 2023. URL: https://redis.com/blog/redisgraph-eol/ (visited on 07/22/2023).

[28] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[29] Chris Lattner et al. "MLIR: scaling compiler infrastructure for domain specific computation". In: *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '21. Virtual Event, Republic of Korea: IEEE Press, Sept. 2021, pp. 2–14. ISBN: 978-1-72818-613-9. DOI: 10.1109/CGO51591.2021.9370308. URL: https://doi.org/10.1109/CGO51591.2021.9370308 (visited on 05/15/2023).

[30] Wilco v. Leeuwen et al. "AvantGraph query processing engine". In: *Proceedings of the VLDB Endowment* 15.12 (Sept. 2022), pp. 3698–3701. ISSN: 2150-8097. DOI: 10.14778/3554821.3554878. URL: https://doi.org/10.14778/3554821.3554878 (visited on 01/26/2023).

[31] Shangyu Luo et al. "Scalable Linear Algebra on a Relational Database System". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. ISSN: 2375-026X. Apr. 2017, pp. 523–534. DOI: 10.1109/ICDE.2017.108.

[32] Hongbin Ma et al. "G-SQL: fast query processing via graph exploration". In: *Proceedings of the VLDB Endowment* 9.12 (Aug. 2016), pp. 900–911. ISSN: 2150-8097. DOI: 10.14778/2994509.2994510. URL: https://dl.acm.org/doi/10.14778/2994509.2994510 (visited on 04/26/2023).

[33] Andrew MacLeod and Martin Sebor. *Atomic/GCCMM/Optimizations - GCC Wiki*. URL: https://gcc.gnu.org/wiki/Atomic/GCCMM/Optimizations (visited on 06/06/2023).

[34] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, June 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: https://dl.acm.org/doi/10.1145/1807167.1807184 (visited on 07/11/2023).

[35] Tim Mattson et al. "LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS". en. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Rio de Janeiro, Brazil: IEEE, May 2019, pp. 276–284. ISBN: 978-1-72813-510-6. DOI: 10.1109/IPDPSW.2019.00053. URL: https://ieeexplore.ieee.org/document/8778338/ (visited on 04/11/2023).

[36] Robert Ryan McCune, Tim Weninger, and Greg Madey. "Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing". In: *ACM Computing Surveys* 48.2 (Oct. 2015), 25:1–25:39. ISSN: 0360-0300. DOI: 10.1145/2818185. URL: https://dl.acm.org/doi/10.1145/2818185 (visited on 07/11/2023).

[37] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! but at what cost?" In: *Proceedings of the 15th USENIX conference on Hot Topics in Operating Systems*. HOTOS'15. USA: USENIX Association, May 2015, p. 14. (Visited on 07/11/2023).

[38] MLIR. *MLIR Open Meeting 2022-01-13: One-Shot Function Bufferization of Tensor Programs*. Jan. 2022. URL: https://www.youtube.com/watch?v=TXEo59CYS9A (visited on 07/23/2023).

[39] Mark Needham and Amy E Hodler. "A comprehensive guide to graph algorithms in neo4j". In: *Neo4j. com* (2018).

[40] *openCypher · openCypher*. URL: https://opencypher.org/ (visited on 05/02/2023).

[41] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2013. ISBN: 978-1-93435-699-9. URL: https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/.

[42] Michel Pelletier et al. "The GraphBLAS in Julia and Python: the PageRank and Triangle Centralities". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. ISSN: 2643-1971. Sept. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622789.

[43] Oskar van Rest et al. "PGQL: a property graph query language". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. GRADES '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 1–6. ISBN: 978-1-4503-4780-8. DOI: 10.1145/2960414.2960421. URL: https://dl.acm.org/doi/10.1145/2960414.2960421 (visited on 07/22/2023).

[44] Martin Rosvall and Carl T. Bergstrom. "Maps of random walks on complex networks reveal community structure". In: *Proceedings of the National Academy of Sciences* 105.4 (Jan. 2008). Publisher: Proceedings of the National Academy of Sciences, pp. 1118–1123. DOI: 10.1073/pnas.0706851105. URL: https://www.pnas.org/doi/10.1073/pnas.0706851105 (visited on 07/26/2023).

[45] Florin Rusu and Yu Cheng. *A Survey on Array Storage, Query Languages, and Systems*. arXiv:1302.0103 [cs]. Feb. 2013. DOI: 10.48550/arXiv.1302.0103. URL: http://arxiv.org/abs/1302.0103 (visited on 07/22/2023).

[46] Yousef Saad. "3. Sparse Matrices". In: *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2003, pp. 73–101. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.ch3. URL: https://epubs.siam.org/doi/10.1137/1.9780898718003.ch3 (visited on 07/19/2023).

[47] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. *Push vs. Pull-Based Loop Fusion in Query Engines*. arXiv:1610.09166 [cs]. Oct. 2016. DOI: 10.48550/arXiv.1610.09166. URL: http://arxiv.org/abs/1610.09166 (visited on 05/13/2023).

[48] Moritz Sichert and Thomas Neumann. "User-defined operators: efficiently integrating custom algorithms into modern databases". In: *Proceedings of the VLDB Endowment* 15.5 (Jan. 2022), pp. 1119–1131. ISSN: 2150-8097. DOI: 10.14778/3510397.3510408. URL: https://dl.acm.org/doi/10.14778/3510397.3510408 (visited on 07/11/2023).

[49] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. 6th ed. New York: McGraw-Hill, 2010. Chap. 13. URL: http://www.db-book.com/.

[50] Upasana Sridhar et al. "Delta-Stepping SSSP: From Vertices and Edges to GraphBLAS Implementations". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 241–250. DOI: 10.1109/IPDPSW.2019.00047.

[51] The LLVM developers. *Re-land "[clang][Interp] Implement C++ Range-for loops" · llvm/llvm-project@2708869*. en. URL: https://github.com/llvm/llvm-project/commit/2708869 (visited on 08/03/2023).

[52] The MLIR developers. *Bufferization - MLIR*. URL: https://mlir.llvm.org/docs/Bufferization/#the-talk (visited on 07/23/2023).

[53] The Neo4J developers. *User-defined procedures - Java Reference*. en. URL: https://neo4j.com/docs/java-reference/5/extending-neo4j/procedures/ (visited on 07/11/2023).

[54] The TigerGraph developers. *tigergraph/gsql-graph-algorithms: GSQL Graph Algorithms*. URL: https://github.com/tigergraph/gsql-graph-algorithms/tree/master (visited on 07/22/2023).

[55] Ruiqin Tian et al. "A High Performance Sparse Tensor Algebra Compiler in MLIR". In: *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. Nov. 2021, pp. 27–38. DOI: 10.1109/LLVMHPC54804.2021.00009.

[56] Michael M. Wolf et al. "Fast linear algebra-based triangle counting with KokkosKernels". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2017, pp. 1–7. DOI: 10.1109/HPEC.2017.8091043.

[57] Milian Wolff. *SDK / Heaptrack · GitLab*. en. July 2023. URL: https://invent.kde.org/sdk/heaptrack (visited on 07/25/2023).

[58] Haoran Xu and Fredrik Kjolstad. "Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (Oct. 2021), 136:1–136:30. DOI: 10.1145/3485513. URL: https://dl.acm.org/doi/10.1145/3485513 (visited on 06/17/2023).

[59] Carl Yang, Aydın Buluç, and John D. Owens. "Implementing Push-Pull Efficiently in GraphBLAS". en. In: *Proceedings of the 47th International Conference on Parallel Processing*. Eugene OR USA: ACM, Aug. 2018, pp. 1–11. ISBN: 978-1-4503-6510-9. DOI: 10.1145/3225058.3225122. URL: https://dl.acm.org/doi/10.1145/3225058.3225122 (visited on 12/23/2022).

[60] Yunming Zhang et al. "GraphIt: a high-performance graph DSL". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), 121:1–121:30. DOI: 10.1145/3276491. URL: http://doi.org/10.1145/3276491 (visited on 10/24/2022).

Acronyms

AST abstract syntax tree

CPU central processing unit

CSE common subexpression elimination

CSR compressed sparse row

DSL domain-specific language

IR intermediate representation

MLIR multi-level intermediate representation

SIMD single instruction, multiple data

SSA single static assignment

Appendix A

Graphalg Implementations of Graph Algorithms

A.1 Benchmark Programs for Interpreter Overhead

The programs listed in this section are equivalent to the original LAGraph implementations (without SuiteSparse extensions).

```
func TriangleCount(graph: Matrix<bool>) -> int {
    L = select(tril, graph, -1);
    U = select(triu, graph, 1);
    C<L, struct> = L (+.one) U.T;

    return reduce(+, C);
}

// SSSP with Delta stepping
func SingleSourceShortestPath(
        graph: Matrix<int>,
        source: index,
        delta: int) -> Vector<int> {
    // Tentative shortest path length
    t = Vector<int>(graph.nrows);
    t[:] = INT64_MAX;
    t[source] = 0;

    tmasked = Vector<int>(graph.nrows);
    tReq = Vector<int>(graph.nrows);
    tless = Vector<bool>(graph.nrows);

    reach = Vector<int>(graph.nrows);
    reach[source] = true;

    s = Vector<int>(graph.nrows);
    s[source] = true;

    AL = select(<=, graph, delta);
    AH = select(>, graph, delta);

    // Used to partially clear other vectors
```

```
    empty = Vector<int>(graph.nrows);

    // while t >= step * Delta not empty
    for step in 0:graph.nrows until reach.nvals == 0 {
        clear s;

        uBound = (step + 1) * delta;

        clear tmasked;
        tmasked<reach> = t;
        tmasked = select(<, tmasked, uBound);

        done = tmasked.nvals == 0;
        for i in 0:graph.nrows until done {
            tReq = tmasked (min.+) AL;
            s<tmasked, struct>[:] = true;
            done = tReq.nvals == 0;
            if !done {
                // Using set intersection
                tless = tReq .< t;

                tless = select(!=, tless, 0);
                done = tless.nvals == 0;
                if !done {
                    reach<tless, struct>[:] = true;

                    clear tmasked;
                    tmasked<tless, struct> = select(<, tReq, uBound);

                    t<tless, struct> = tReq;
                    done = tmasked.nvals == 0;
                }
            }
        }

        clear tmasked;
        tmasked<s, struct> = t;

        tReq = tmasked (min.+) AH;

        // Using set intersection
        tless = tReq .< t;

        t<tless> = tReq;

        reach<tless>[:] = true;

        reach<s, struct> = empty;
    }

    return t;
}
```

```
func PageRank(graph: Matrix<real>,
              graph_T: Matrix<real>,
              d_out: Vector<int>,
              damping: real,
              tol: real,
              itermax: int) -> Vector<real> {
    n = graph.nrows;
    teleport = (1 - damping) / n;
    rdiff = 1.0;

    t = Vector<real>(n);
    r = Vector<real>(n);
    w = Vector<real>(n);

    r[:] = 1.0 / n;

    d = apply(/, d_out, damping);

    d1 = Vector<real>(n);
    d1[:] = 1.0 / damping;
    d = add(max, d1, d);

    for i in 0:itermax until rdiff <= tol {
        swap r t;

        w = t ./ d;

        r[:] = teleport;
        r += graph_T (+.second) w;

        t -= r;
        t = apply(abs, t);
        rdiff = reduce(+, t);
    }

    return r;
}

// BFS with push-pull optimization
func BreadthFirstSearch(
        graph: Matrix<bool>,
        graph_t: Matrix<bool>,
        out_degree: Vector<int>,
        source: index) -> Vector<int> {
    n = graph.nrows;

    pi = Vector<int>(n);
    pi[source] = int(source);
    q = Vector<int>(n);
    q[source] = int(source);

    w = Vector<int>(n);
```

```
nq = index(1);
alpha = 8.0;
beta1 = 8.0;
beta2 = 512.0;

n_over_beta1 = int(n / beta1);
n_over_beta2 = int(n / beta2);

do_push = true;
last_nq = index(0);
edges_unexplored = graph.nvals;
any_pull = false;

push_pull = true;
nvisited = 1;
for k in 1:n until nvisited >= n {
    if push_pull {
        if do_push {
            growing = nq > last_nq;
            switch_to_pull = false;
            if edges_unexplored < n {
                push_pull = false;
            } else if any_pull {
                switch_to_pull = growing && (nq > n_over_beta1);
            } else {
                w<q, replace, struct> = out_degree;
                edges_in_frontier = reduce(+, w);
                switch_to_pull = growing
                    && (edges_in_frontier > (edges_unexplored / alpha));
            }
            if switch_to_pull {
                do_push = false;
            }
        } else {
            shrinking = nq < last_nq;
            if shrinking && (nq <= n_over_beta2) {
                do_push = true;
            }
        }
        any_pull = any_pull || (!do_push);
    }

    if do_push {
        q<!pi, replace, struct> = q (any.secondi) graph;
    } else {
        q<!pi, replace, struct> = graph_t (any.secondi) q;
    }

    last_nq = nq;
    nq = q.nvals;
```

```
        pi<q, struct> = q;

        nvisited = nvisited + nq;
    }

    return pi;
}
```

A.2 Benchmark Programs for Comparing Backends

The TriangleCount the PageRank implementations here match the ones presented in Section A.1. The only difference is that we compute the transpose, out-degree etc. inside the algorithm rather than passing them as parameters, since the database does not cache them. SingleSourceShortestPath and BreadthFirstSearch are simplified to remove conditional branching.

```
call(
    "
        func TriangleCount(graph: Matrix<bool>) -> int {
            L = select(tril, graph, -1);
            U = select(triu, graph, 1);
            C<L, struct> = L (+.one) U.T;

            return reduce(+, C);
        }
    ",
    (%val) = "TriangleCount" (
        matrix(
            projection(
                access(%0, "friend"), {
                    %row = src(%0),
                    %col = trg(%0),
                    %val = %0.weight,
                }
            ),
            %row, %col, %val),
    ),
)

call(
    "
        func SSSP(graph: Matrix<int>, source: index) -> Vector<int> {
            v = Vector<int>(graph.nrows);
            v[source] = 0;

            for i in 0:300 {
                v min= v (min.+) graph;
            }

            return v;
        }
    ",
```

```
    (%row, %val) = "SSSP" (
        matrix(
            projection(
                access(%0, "friend"), {
                    %row = src(%0),
                    %col = trg(%0),
                    %val = %0.weight,
                }
            ),
            %row, %col, %val),
        v:START,
    ),
)

call(
    "
        func PageRank(graph: Matrix<bool>,
                    damping: real,
                    tol: real,
                    itermax: int) -> Vector<real> {
            cnt = apply(one, graph);
            d_out = reduceRows(+, cnt);

            n = graph.nrows;
            teleport = (1 - damping) / n;
            rdiff = 1.0;

            t = Vector<real>(n);
            r = Vector<real>(n);
            w = Vector<real>(n);

            r[:] = 1.0 / n;

            d = apply(/, d_out, damping);

            d1 = Vector<real>(n);
            d1[:] = 1.0 / damping;
            d = add(max, d1, d);

            for i in 0:itermax until rdiff <= tol {
                swap r t;

                w = t ./ d;

                r[:] = teleport;
                r += graph.T (+.second) w;

                t -= r;
                t = apply(abs, t);
                rdiff = reduce(+, t);
            }
```

```
            return r;
        }
    ",
    (%row, %val) = "PageRank" (
        matrix(
            projection(
                access(%0, "friend"), {
                    %row = src(%0),
                    %col = trg(%0),
                    %val = %0.weight,
                }
            ),
            %row, %col, %val),
        0.85,
        0.0001,
        100
    ),
)

call(
    "
        func BreadthFirstSearch(graph: Matrix<bool>, source: index) -> Vector<int> {
            n = graph.nrows;
            out_degree = reduceRows(+, graph);

            pi = Vector<int>(n);
            pi[source] = int(source);
            q = Vector<int>(n);
            q[source] = int(source);

            nq = index(1);

            nvisited = 1;
            for k in 1:n until (nq == 0) || (nvisited >= n) {
                q<!pi, replace, struct> = q (any.secondi) graph;

                nq = q.nvals;

                pi<q, struct> = q;

                nvisited = nvisited + nq;
            }

            return pi;
        }
    ",
    (%row,%val) = "BreadthFirstSearch" (
        matrix(
            projection(
                access(%0, "friend"), {
                    %row = src(%0),
                    %col = trg(%0),
```

```
                    %val = %0.weight,
                }
            ),
            %row, %col, %val),
            v:START,
    ),
)
```

A.3 Implementations of Betweenness Centrality and Connected Components

The implementation of Connected Components is based on the simple component-at-a-time algorithm. For Betweenness Centrality we base our implementation on LAGraph, but substitute C arrays for submatrix indexing.

```
func ConnectedComponents(graph: Matrix<bool>) -> Vector<int> {
    label = Vector<int>(graph.nrows);
    label[:] = 0;
    reach = Vector<bool>(graph.nrows);
    front = Vector<bool>(graph.nrows);

    cid = 1;
    unlabeled = int(graph.nrows);
    vid = index(0);
    for v in 0:graph.nrows until unlabeled == 0 {
        vid = v;
        if label[vid] == 0 {
            // Vertex is not part of a known component.
            // Find all nodes reachable from this vertex (the new component).
            clear reach;
            reach[vid] = true;
            clear front;
            front[vid] = true;
            for unused in 0:graph.nrows until (front.nvals == 0) || (reach.nvals == unlabeled) {
                front<!reach, replace, struct> = front (any.one) graph;
                reach<front, struct> = true;
            }
            // Label all the vertices we found
            label<reach, struct>[:] = cid;
            unlabeled = unlabeled - reach.nvals;
            cid = cid + 1;
        }
    }

    return label;
}

func Betweenness(
        graph: Matrix<bool>,
        graph_t: Matrix<bool>,
        sources: Vector<int>) -> Vector<real> {
```

```
n = graph.nrows;
ns = sources.nrows;

paths = Matrix<real>(ns, n);
frontier = Matrix<real>(ns, n);

for i in 0:ns {
    src = sources[i];
    paths[i, src] = 1.0;
    frontier[i, src] = 1.0;
}

frontier<!paths, replace, struct> = frontier (+.first) graph;
//S = [Matrix<bool>(ns, n); n+1];
S = Matrix<bool>(ns*(n+1), n);
Sd = Matrix<bool>(ns, n);


// BFS stage
last_was_pull = false;
frontier_size = frontier.nvals;
depth = index(0);
for d in 0:n until frontier_size <= 0 {
    // Sd = S[depth]
    //Sd = S[depth*ns:(depth+1)*ns, :];
    Sd<frontier, replace, struct> = true;
    // S[depth] = Sd;
    S[depth*ns:(depth+1)*ns] = Sd;

    paths += frontier;

    pull_min_density = 0.10;
    if last_was_pull {
        pull_min_density = 0.06;
    }
    frontier_density = real(int(frontier_size)) / (ns*n);
    do_pull = frontier_density > pull_min_density;

    if do_pull {
        frontier<!paths, replace, struct> = frontier (+.first) graph_t.T;
    } else {
        frontier<!paths, replace, struct> = frontier (+.first) graph;
    }

    last_was_pull = do_pull;
    frontier_size = frontier.nvals;
    depth = index(depth + 1);
}

// Betweenness centrality phase
bc_update = Matrix<real>(ns, n);
bc_update[:, :] = 1;
```

```
    W = Matrix<real>(ns, n);
    for d in 1:depth {
        i = depth-d;
        // Sd = S[i]
        Sd = S[i*ns:(i+1)*ns, :];
        W<Sd, replace, struct> = bc_update ./ paths;

        wsize = W.nvals;
        // Sd = S[i-1]
        Sd = S[(i-1)*ns:i*ns, :];
        ssize = Sd.nvals;
        w_density = real(int(wsize)) / (ns*n);
        w_to_s_ratio = real(int(wsize)) / ssize;
        do_pull = ((w_density > 0.1)  && (w_to_s_ratio > 1))
               || ((w_density > 0.01) && (w_to_s_ratio > 10));

        if do_pull {
            W<Sd, replace, struct> = W (+.first) graph.T;
        } else {
            W<Sd, replace, struct> = W (+.first) graph_t;
        }

        bc_update += W .* paths;
    }

    centrality = Vector<real>(n);
    centrality[:] = -int(ns);
    centrality += reduceRows(+, bc_update.T);
    return centrality;
}
```