GSTACO: A Generalized Sparse Tensor Algebra Compiler

by

Alexandra Dima

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author . . . .
Department of Electrical Engineering and Computer Science
January 20, 2023

Certified by . . . .
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

[Page intentionally left blank]

GSTACO: A Generalized Sparse Tensor Algebra Compiler

by

Alexandra Dima

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Many applications in engineering and computer science are characterized by sparse multi-dimensional data. Therefore, optimizations for sparse tensor algebra have received a lot of attention lately. Several hardware and software solutions have emerged in order to speed up the computation of certain tensor expressions, but none of them provides an interface that is general and comprehensive enough to meet the requirements of complex applications like graph analysis. In this work we attempt to identify where previous solutions fell short and build the Generalized Sparse Tensor Algebra Compiler (GSTACO), a new compiler aiming to fill in the engineering gaps of efficient sparse computation.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

[Page intentionally left blank]

Acknowledgments

I would like to thank professor Saman Amarasinghe for welcoming me into his research group, Compilers At MIT (COMMIT), and for providing continuous guidance and support throughout my time building GSTACO. Additionally, I am grateful for the mentorship of Ajay Brahmakshatriya, who first introduced me to the idea of encoding graph algorithms in tensor algebra and who offered most valuable advice in compiler and performance engineering. Willow Ahrens, author of Finch - the sparsity generator underneath GSTACO, also played an instrumental role by addressing all my feature requests and Github issues as well as helping me understand efficient sparsity code generation. I found our discussions highly motivating and productive. Outside of MIT, I am very thankful to my fiancee who listened to my research monologues and inspired me to study C++ in greater depth, which was immensely helpful in the completion of this thesis. Lastly, I owe many thanks to my family and friends for their support throughout my 5-year journey at MIT.

[Page intentionally left blank]

Contents

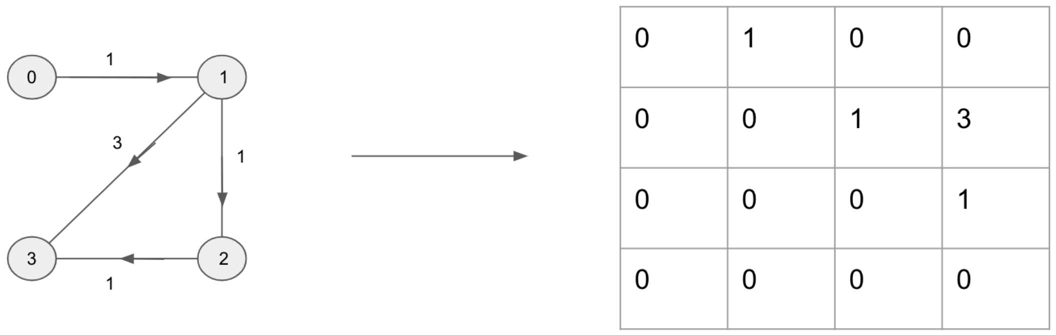
List of Figures

Chapter 1

Introduction

1.1 Background

Tensor algebra and decomposition have become prevalent in today's technological space, from numerical linear algebra, statistical models [21] and machine learning frameworks[2] to signal processing, computer vision and graph analysis [14]. When expressed in tensor algebra, many of these applications actually end up operating on sparse tensors, which are multi-dimensional arrays containing a lot of zero entries. This property enables us to save on computation time by developing algorithms for addition and multiplication which skip those entries that would not otherwise change the final result of the computation. There are also space optimizations that can be performed when sparsity is assumed, like only storing the non-zero elements in memory thus leading to a compressed version of the tensor. However, these algorithms can be difficult to implement by the programmer, so there is a need for support from the compiler. The current state-of-the-art work in this area is The Tensor Algebra Compiler, a DSL (Domain Specific Language) as well as C++ framework which can efficiently compute a wide range of tensor expressions, from simple sparse matrix-vector and matrix-matrix multiplications (SpMV and SpMM) to the more complex matricized tensor times Khatri-Rao product (MTTKRP). [13]



0	1	0	0
0	0	1	3
0	0	0	1
0	0	0	0

1.2 Motivation

However, despite its high performance on sequential tensor expressions, TACO fails to meet all the requirements that most tensor algebra applications have. A particularly compelling example are graph algorithms like Pagerank, BFS [18], Single Source Shortest Path (SSSP) [6], Connected Components (CC), Betweenness Centrality (BC) [9] and many others. Graphs can be represented in linear algebra as adjacency matrices, where the element at position (i,j) coincides with the weight of the edge between nodes i and j. For instance, figure 1.2 shows a 4-node directed graph which can be represented as a 4x4 matrix with non-zero values at indices (0,1), (1,2), (1,3), and (2,3). This graph is relatively sparse, with many of the entries in the adjacency matrix being zero. This is often the case with real world data like social networks or road networks. No one is friends with almost everyone just as no city has direct roads to almost all other cities in the country. Therefore, state-of-the-art sparsity compilers like TACO should be a good fit for these kinds of problems.

This is not, however, the case. Simply compiling a series of TACO statements and embedding the result into a hand-written C/C++ program does not reach the desired performance. There are a number of reasons for this limitation.

First of all, TACO has inadequate support for multi statements, which represent computation kernels that produce multiple tensors from the same loop. More specifically, when two kernels contain a common subexpression, it is wasteful to compute them in two individual loops. A tensor compiler should be expected to compute the common subexpression once and only generate new loops for the non-overlapping parts of the tensor definitions. The presence of reductions makes this a more nuanced task than merely performing loop fusion, which is what TACO does currently.

Secondly, TACO only supports fixed sparsity, meaning that once a tensor has been defined to be either sparse or dense, that characteristic can not change. In contrast, the sparsity of the graph data structures we are working with can change throughout the course of the algorithm. For example, in BFS we start with a mostly empty array of parent pointers and add non-zero entries along the way. The format of the tensor therefore changes from sparse to dense as we discover more and more of the graph. TACO has no way of detecting that in order to change its optimization approach.

Thirdly, TACO does not have any support for control flow like looping or branching constructs, which are extensively used as part of the inherently iterative algorithms that apply to graphs. For example, BFS maintains a working queue of nodes to process and performs tensor operations for each of them. Since this queue has variable length, there is an unknown number of TACO invocations that may need to happen until the problem is solved. Without outer loop functionality, TACO needs to rely on the C++ 'for' construct in which case it would miss many opportunities for sparsity optimization. Similarly, running Pagerank requires performing an if-else check inside the tensor-computing loop in order to avoid division-by-zero errors. TACO does not support this feature out of the box and it would take a significant user effort to encode the if-else functionality as a custom TACO operator.

Lastly, TACO only supports addition as a dimension reduction operation. However, in order to solve SSSP we also need a way to extract the minimum element along a tensor dimension. Similarly, other problems require reduction operators like OR and CHOOSE, which are not supported in TACO either.

There are also some application-specific features that TACO does not handle, like tensors acting as priority queues (needed in SSSP) and Scatter outputs (needed in CC).

1.3 Contribution

In light of the above limitations, we designed the Generalized Sparse Tensor Algebra Compiler (GSTACO), which is a DSL with a stand-alone compiler implemented in C++ that can address most of TACO's missing functionality. This new language is able to support the implementation of a diverse set of graph algorithms, including Pagerank, BFS, SSSP, BC, and CC and is general enough to potentially map other tensor algebra applications as well.

We break this work into two contributions:

1.3.1 Concept

We began by defining a highly expressive and adaptable syntax for our DSL. The front-end is a functional language which encodes all tensor expressions in the Einsum notation [7]. However, we add four extensions to this notation in order to support the functionality missing in TACO:

1. new reduction operators - MIN, CHOOSE, OR - which allow expressing more kinds of computation
2. cross-kernel loops represented by the star function operator, which enable additional optimizations
3. scheduling annotations, which can express iteration order and dynamic tensor format changes across iterations
4. statements that produce multiple outputs in order to avoid wasteful computation

1.3.2 Implementation

We then built GSTACO, an end-to-end compiler for the proposed DSL which emits C++ as well as C-embedded Julia code for CPU. We build on top of another tensor compiler called Finch which takes care of complex tensor expressions (kernels), while

GSTACO fills the cross-kernel optimization gap. Our framework consists of a parser, a high-level IR (Intermediate Representation), a lowering framework as well as code generation. We also perform a data-flow optimization for memory reuse, which proves the feasibility of cross-kernel optimizations in the context of our language.

Chapter 2

Related Work

GSTACO is of course standing on the shoulders of giants as there have been many previous contributions, in hardware and software alike, to the problem of efficient tensor computation under sparsity. Besides TACO, whose shortcomings were the main drive for this work, we also acknowledge projects like Finch and Graphit. The former is actively built upon by GSTACO while the latter served as baseline for GSTACO functionality and performance. This section will briefly discuss each of these as well as other related works and how they differ from GSTACO.

2.1 Finch

Finch is a very recent sparsity compiler based on Looplets [3]. It uses iteration protocols, which is a novel abstraction over structured arrays that enables easier coiteration over different sparsity patterns. This new approach makes Finch a more feasible solution than TACO, as it supports more reduction operators and multiple outputs. However, Finch is still only concerned with single tensor expressions and is therefore missing out on cross-kernel optimizations and outer loop scheduling. Nonetheless, we identified Finch to be a very potent sparsity generator, so we decided to build GSTACO as a wrapper around Finch. Finch is invoked for each tensor expression requiring advanced computation, while GSTACO is taking care of the efficient context switching in-between.

2.2 Graphit

GraphIt[22] is a recent state-of-the-art DSL for fast graph computations on inputs of various sizes. It takes a completely different approach since it does not attempt to translate graph problems to a set of tensor computations. It has its own optimization stack which includes techniques like reducing instruction counts, parallelizing loops, and improving data locality. In comparison to Graphit, we don't restrict ourselves to just graphs, and instead target any problem that can be formulated as a tensor algebra problem. Nonetheless, GraphIt served as both a proof of concept as well as a baseline for performance evaluation for GSTACO. We used all the graph algorithms officially tested by Graphit to guide some of our design choices.

2.3 GraphBLAS

Another relevant piece of work is the GraphBLAS library [1], which provides an API for implementing graph algorithms as matrix computations. Just as GSTACO, it was inspired by the crucial observation that graphs are basically sparse matrices and that one iteration of BFS can be formulated as a sparse matrix-vector multiplication [1]. GraphBLAS specifies useful building blocks for graph applications, but putting those building blocks together and optimizing the end result is still the user's job. GSTACO's syntax provides a higher level of abstraction and therefore requires much less user effort. Additionally, the primitives in GraphBLAS (mainly sprase matrix, sparse vector, and operators), do not appear to solve the problem of more complex data types required by graph applications, like priority queues, which are sparse across multiple dimensions.

2.4 Others

Additional contributions that inspired our work are ExTensor, Taichi, and COMET.

ExTensor [10] is a recent accelerator which produces good performance by only sending non-zero tensor entries to the arithmetic unit within hardware. However, being bound to a specific hardware architecture makes this approach much less portable and adaptable than a software solution. Additionally, ExTensor is missing some optimizations that could be made at the software level, like choosing among the many possible storage formats for sparse tensors (COO, CSR, CSF). Our Finch-based solution does not have these limitations, as Finch supports a variety of formats and GSTACO switches between them according to a user-specified schedule.

From the software side, Taichi [11] is another DSL with an optimizing compiler for sparse data structures, but it is specifically targeted towards physical simulation and rendering applications. Our ambition is to build a more general framework that can map well to any problem involving sparse tensors.

COMET[20] is a DSL very similar to TACO in terms of approach and performance, and therefore has many of the same limitations. It only supports compiling individual tensor expressions, without any support for outer loops or changing sparsity.

Chapter 3

GSTACO Language

Having introduced GSTACO at a high-level, we will now go more in depth into how it works. This chapter describes the GSTACO language constructs which enable important features for tensor-based program and provides examples from real GSTACO programs for graph algorithms.

3.1 Syntax & Semantics

Our language takes inspiration from the Einsum notation [7], which is a compressed way of expressing reduction by summation over the elements of a collection. For example, a matrix-vector multiplication operation can be briefly described in Einsum as:

$$a_i = \sum_{j=1}^{N} b_{ij} * v_j \tag{3.1}$$

In GSTACO, variables a, b, and v would be declared as tensors of integers of appropriate dimensions and the kernel would translate to:

$$a[i] = b[i][j] * v[j] | j : (+, 0) \tag{3.2}$$

The pipe is used to mark the presence of a reduction, which means that there are more index variables on the righthand side than on the lefthand side of the tensor expression, and hence these need to be reduced/accumulated using the plus operator. The pipe is then followed by specifications (operator and initial value) for each of the accumulated index variables. Notice how GSTACO is actually more powerful than the classic Einsum notation as it allows users to customize the operator as well as the initial value. However, diverse reduction operators are not the only extension to Einsum. In order to fulfill all the requirements of a tensor algebra language, we also added constructs like types, functions, the Star(*) operator for outer loops, tensor storage formats, as well as simple but specific scheduling commands. Each of these will be presented in detail.

Additionally, it is worth mentioning that GSTACO is a functional language, with immutable variables and functions that only create and return new values computed from old values. The motivation behind this design decision was making GSTACO easily extendable to more back-ends other than CPU. For example, next-generation architectures like SWARM[12] break code down into potentially parallelizable chunks called tasks and attempt to execute them in parallel for increased performance. These architectures detect any data dependencies during the execution of parallel tasks in order to abort/restart the dependent tasks and preserve correctness. This means that efficient SWARM code would ideally contain very few data dependencies so that the code can benefit from the high degree of parallelization. Imposing the constraint that all variables are immutable offers the guarantee that they will not be both read and written by two parallel tasks. Hence, this makes it possible to schedule many computations simultaneously, provided that their inputs become available at the same time. Figures 3.1 and 3.1 show examples of code that would achieve different performance on a Swarm architecture.

$$\mathrm{a}_i = a_{i+1}$$

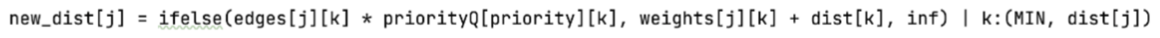
Figure 3-1: performs poorly on Swarm

$$\mathrm{b}_i = a_{i+1}$$

Figure 3-2: maps well to efficient Swarm code

3.1.1 Reduction Operators

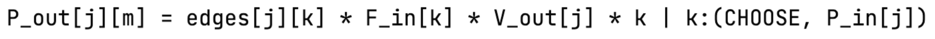
Many graph applications require accumulating values with MIN, OR, and CHOOSE operators. For example, in SSSP we pop nodes from the queue and relax their outgoing edges in order to make progress towards discovering new shortest paths. When we try to express this in Einsum, we realize that summation is not enough; we need to take the minimum over all relaxed edges from a node. This can be achieved with the syntax in figure 3-3.



```
new_dist[j] = ifelse(edges[j][k] * priorityQ[priority][k], weights[j][k] + dist[k], inf) | k:(MIN, dist[j])
```

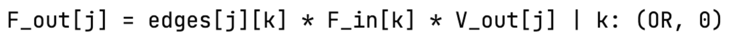
Figure 3-3: SSSP: computing new distances based on the nodes at the current priority and edge weights

Another common example is BFS, where we assign a parent to an unvisited node once we discover it through any of its neighbours at the current level. We can express "any of" through a CHOOSE/OR reduction over the new level/frontier computation, as shown in figures 3-4 and 3-5. The CHOOSE operator returns the first non-zero element from a collection. As such, we use CHOOSE in order to assign a parents to the visited nodes. Similarly, we use OR in order to compute membership in the next level/frontier. Under the hood, both reductions should be implemented with early break out of the iteration once a non-zero/one is found.



```
P_out[j][m] = edges[j][k] * F_in[k] * V_out[j] * k | k:(CHOOSE, P_in[j])
```

Figure 3-4: BFS: computing the parents of the nodes in the next frontier



```
F_out[j] = edges[j][k] * F_in[k] * V_out[j] | k: (OR, 0)
```

Figure 3-5: BFS: computing the next frontier of nodes from the old frontier, edges adjacency matrix and visited information

3.1.2 Tensor Variables and Types

In GSTACO, we have both scalar and tensor variables as well as 4 builtin types: Int, Float, Bool, and tensor types written as "P[N1][N2]...[Nx]". Here, P is the primitive type of the elements in the tensor and N1-Nx are the dimensions of the tensor. Variables need to be declared as either global variables or function inputs/outputs before being defined. Additionally, tensor variables may only be defined once after declaration and never mutated, just like in other functional languages.

```
edges int[Dense[N]][SparseList[N]]
```

Figure 3-6: Declaring 'edges' as a CSR tensor

3.1.3 Functions

Since GSTACO has a functional layout, functions build their outputs as immutable objects, and can use as many temporary variables as needed in order to express complex computations. Syntactically, functions always starts with the "Let" keyword and list the names and types of all inputs as well as outputs, after the arrow. This is illustrated in figure 3-7. Additionally, functions don't merely provide the user with a means for reusing code, but they are also heavily used in conjunction with the star operator described in the next section.

```
Let Init(source int) -> (dist float[N], priorityQ int[P][SparseList[N]])
    dist[j] = ifelse(j != source, P, 0)
    priorityQ[p][j] = (p == 1 && j == source) || (p == P && j != source)
End
```

Figure 3-7: Function declaration for initializing tensors in SSSP

3.1.4 Star Operator

The main contribution behind GSTACO is being able to efficiently implement entire tensor applications, not only individual kernels. It was therefore crucial to design the language and intermediary representation in a way that is conducive to identifying opportunities for optimization across kernels. We also made the observation that many algorithms require computing the same tensor expressions repeatedly until a certain state or steady state is reached. For example, in BFS we keep visiting neighbouring nodes until all nodes in the graph have been visited. Hence, we introduced the concept of an outer loop - the repeated execution of a function computing one or multiple kernels by passing the output back as inputs to the function until a specified end condition becomes true. This behavior is encapsulated by the star operator for function calls.

```
_, P_out, _, _ = BFS_Step*(F_in, P_in, V, temp_in) | (#1 == 0)
```

Figure 3-8: BFS: repeatedly visiting levels of equidistant nodes from the source until no more unvisited nodes can be reached

As shown in figure 3-8, the Star function call defines its end condition after the pipe symbol. This condition is an expression which evaluates to a boolean, and more specifically, it should eventually evaluate to true in order to ensure the termination of the program. It is most often written in terms of one or more of the function outputs which are referenced by their absolute position in the source code. Here, in BFS, the ending condition is satisfied when the most recently computed frontier becomes empty (no more nodes to visit). Since the frontier is the first argument to the function and the first output, it will be represented with the syntax "#1" in the condition. The underscores on the left hand side act as wildcards and can be useful when we don't necessarily care about some of the outputs beyond the iteration.

Similarly, in SSSP we keep relaxing edges from the nodes with the currently lowest priority from the queue, until no more nodes with that priority exist. The end condition here is written in terms of both the priority queue argument and the current priority level argument, as in figure 3-9.

```
new_dist, new_priorityQ, _, temp_out = UpdateEdges*(dist, priorityQ, priority, temp_in) | (#2[#3] == 0)
```

Figure 3-9: SSSP: process all nodes from the queue with given priority level

Finally, it should be noted that in order for the Star operator to be applied, the function needs to have the same number of inputs and outputs, otherwise the behavior is not defined.

3.1.5 Tensor Formats

There is, however, one more layer of complexity over tensor abstractions, and that has to do with specifying storage formats. GSTACO adopts a storage scheme similar to TACO's level formats[13], where each dimension of a tensor can be declared as either dense or compressed. The syntax "P[N1][N2]...[Nx]" makes all dimensions Dense by default, while "P[SparseList[Ni]]..." would be used to indicate sparsity. For example, figure 3-6 shows the "edges" tensor in PageRank is declared as having a dense and a compressed dimension, which is equivalent to the known CSR formats[5]. This format is likely to be optimal in many programs since most graphs used in real life are sparse. On the other hand, the ranks tensor in PageRank is dense since this is not expected to contain a lot of zeroes. An important thing to note is that these formats are not static. They may change throughout the execution of an algorithm, and so they need to be adjusted. More specifically, while executing an outer loop, one of the input tensors to the function may start out as empty and end up filled with values, or the other way around. This means that using only one format is not efficient. The better approach is to start out sparse and dynamically "densify" at some point in the execution. Exactly when this should happen varies from case to case, and should be specified by the user with the scheduling constructs described next.

3.1.6 Scheduling: Format Switch

As mentioned before, formats are a dynamic property and they need to be flexible enough to allow efficient storage all throughout the execution. Therefore, GSTACO grants users the ability to dictate when a format switch should happen for tensors whose properties change across the outer loop. They can do so by inserting one or more `FormatRule` annotations above outer loop definitions, as pictured in BFS 3-10. Each rule specifies the index in the argument list of the underlying function call which identifies the tensor changing formats. Besides, the rules also include the starting and ending format, as well as an expression which is the transition condition. For the BFS example, we switch the visited tensor from sparse to dense once a quarter of all nodes have already been visited. Similarly, the frontier tensor starts sparse and becomes dense when the number of frontier nodes is larger than a fifth of all nodes. However, the size of the frontier does not vary monotonically, and so it is possible for it to pass the $\frac{N}{5}$ threshold multiple times. In the case when the size decreases below this number, the format switch will take place in reverse (from dense to sparse).

```
FormatRule #3 int[SparseList[N]] -> int[N] @ (#5 > N / 4)
FormatRule #1 int[SparseList[N]] -> int[N] @ (#6 > N / 5)
_, P_out, _, _, _, _ = BFS_Step*(F_in, P_in, V, temp_in, V_size, F_size) | (#1 == 0)
```

Figure 3-10: BFS loop optimized for tensor formats. Two more inputs have been added in order to keep track of the metrics that drive the format switch decision (number of visited and frontier nodes)

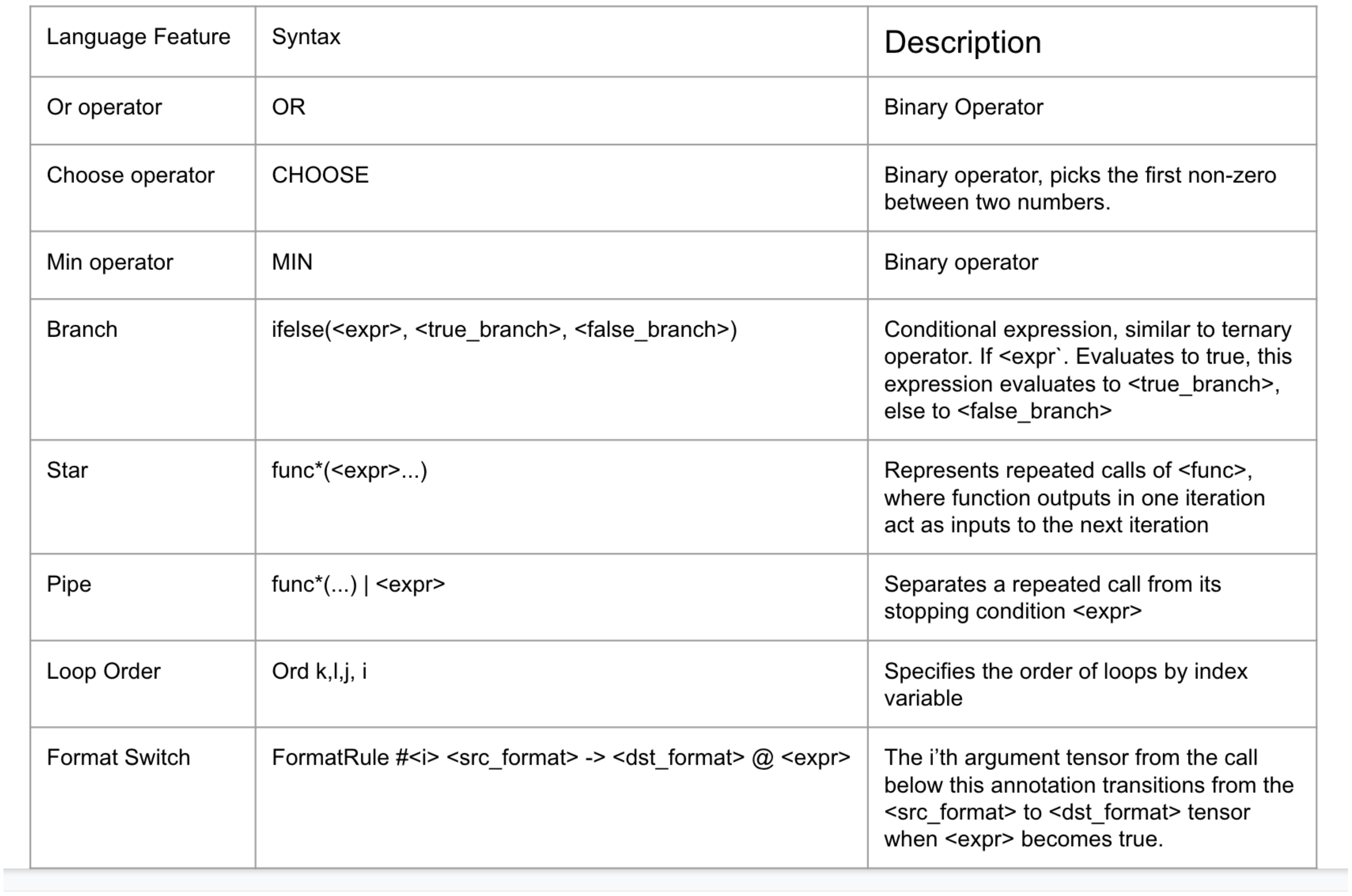
3.1.7 Scheduling: Loop Ordering

Another feature which conceptually pertains to the schedule rather than to the tensor algorithms themselves is the iteration order of nested loops. The importance of this aspect is best highlighted by the out-degree tensor in PageRank. In order to compute the out-degrees of all nodes, the kernel might iterate thorough all nodes, and for each of them iterate through all its out-going edges, reducing the results into one element of the ranks tensor. Alternatively, for each of the nodes we can also iterate through all its in-coming edges and update different rank locations. Which approach is more efficient depends on the format of the edges. We can easily switch between them by specifying a different i-j order right after the stop condition of a function star operator, as seen in figure 3-11. The default behavior is when no orders are given, in which case the loop order will be the order in which index variables first show up in the tensor definition, from left to right.

```
out_d[j] = edges[i][j] | i:(+, 0) Ord i, j
```

Figure 3-11: Pagernk: by default, the loop order of this kernel would be j, i. However, out-degree requires reordering the loops in order to optimize access over the sparse tensor 'edges'

Lastly, a summary of the most important language features presented so far can be found in table 3-12



Language Feature	Syntax	Description
Or operator	OR	Binary Operator
Choose operator	CHOOSE	Binary operator, picks the first non-zero between two numbers.
Min operator	MIN	Binary operator
Branch	ifelse(<expr>, <true_branch>, <false_branch>)	Conditional expression, similar to ternary operator. If <expr`. Evaluates to true, this expression evaluates to <true_branch>, else to <false_branch>
Star	func*(<expr>...)	Represents repeated calls of <func>, where function outputs in one iteration act as inputs to the next iteration
Pipe	func*(...) | <expr>	Separates a repeated call from its stopping condition <expr>
Loop Order	Ord k,l,j, i	Specifies the order of loops by index variable
Format Switch	FormatRule #<i> <src_format> -> <dst_format> @ <expr>	The i'th argument tensor from the call below this annotation transitions from the <src_format> to <dst_format> tensor when <expr> becomes true.

Figure 3-12: Language Feature Summary

Chapter 4

System Overview & Implementation

GSTACO is both a research and engineering effort dedicated to improving the way that state of the art sparsity tools are used. Details of implementation deserve significant word count and, as such, this chapter outlines the most important technical components one has to familiarize themselves with in order to produce further work on GSTACO. Figure 4-1 depicts the different stages of GSTACO as a system. Firstly, the syntax described in chapter 3 is passed through a lexer and parser in order to produce the abstract syntax tree (AST). Afterwards, we perform some cleanup passes over the AST in order to address some of the shortcomings of a Bison-generated parser, though these may not be needed if a custom, hand-written parser would be produced in the future. The resulting high level intermediary representation (IR) is fed into a pipeline of visitors and rewriters for optimization and lowering to a low-level IR. Lastly, we have a code generation phase which visits each node in the low level IR and emits C++ code embedding some Julia snippets.

This chapter will also go through the two different approaches we tried for generating sparse code (TACO vs Finch), as well as the details of why we chose to build upon the Finch compiler. We offer an analysis of Finch advantages and disadvantages over TACO and describe solutions to the challenges we encountered during the integration between GSTACO and Finch.

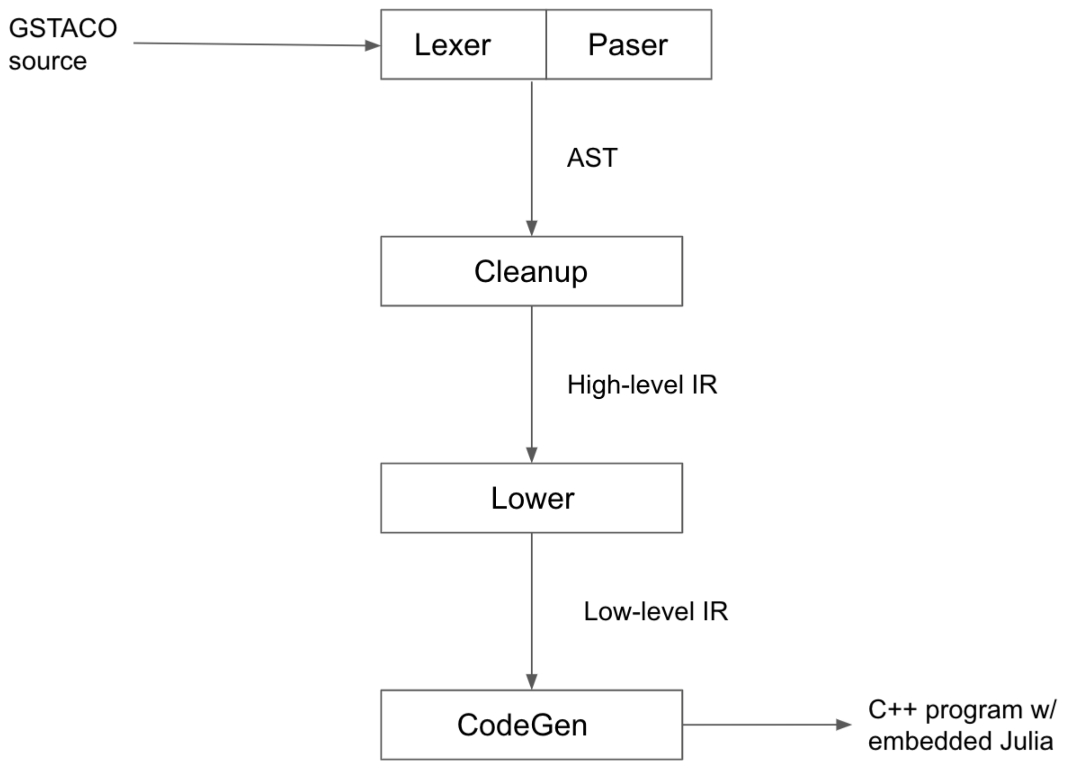


Figure 4-1: System Diagram

4.1 Intermediate Representation

The second step after designing the GSTACO language was to pin down the intermediary representation (referred to as IR in this document). The IR could not simply map one to one with the language syntax since that would make the job of lowering to C++ much harder. It also should not be so complex that its constructs are lower level than C++ itself, which is the product of the compilation in our case. Therefore, any LLVM-like representation optimized to generating low level assembly was unnecessary and suboptimal. We therefore chose a middle ground, where we designed our own hierarchy of C++ classes to serve as a base for program analysis.

4.1.1 Module

The top-level node of the IR is called Module, and it represents the entire GSTACO program read from one file. The module contains one or more module components, each of which can be a function declaration (FuncDecl), or a tensor variable initialization (Initialize). The initialization has a field for the tensor variable that it initializes (TensorVar). At the same time, the tensor variable has not only a string name, but also a type of class TensorType. We show a tree representation of the data structures in figure 4-2, where green arrows represent class inheritence relationships, red arrows are for class fields, and square brackets around nodes represent an array of nodes of the specified type.

4.1.2 Function Declaraion

The function declaration node has several child nodes corresponding to its inputs, outputs, storages for outputs, as well as the body of the function, as seen in figure 4-3. Each of these fields is an array containing other IR nodes. For example, the body of the function consists of an array of Statements, which are a type of valid constructs within a function. The reader might now be wondering what is the purpose of the output-storage field. Behind the scenes, GSTACO maintains the contract that the caller of a function allocates memory for the results and the callee builds its outputs

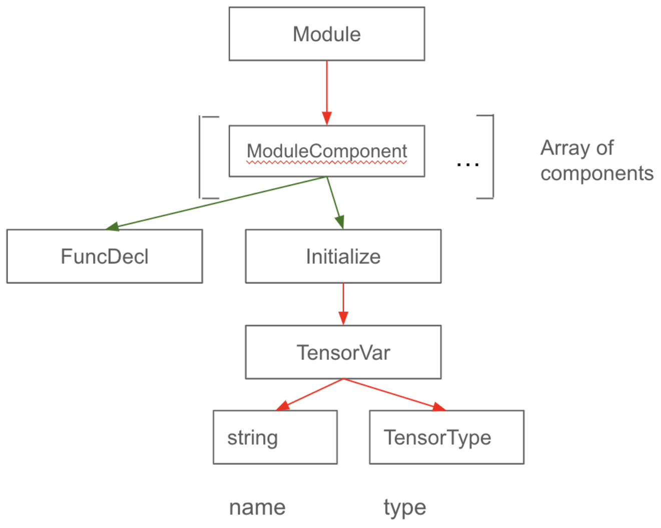


Figure 4-2: Module Components

following the pointers passed in by the caller. Therefore, each function output that is a tensor and not a primitive type will require an associated input for the storage. This is an easy way of ensuring that memory is only allocated once and it also enables certain memory reuse optimizations (to be described later). One exception to this memory management strategy is the Main function (mandatory entry point of the program) which does not take any arguments and, instead, allocates all the memory required for outputs or other functions that it calls.

4.1.3 Statements

There are 3 subtypes to the Statement node: Initialize (already covred), Allocate, and Definition.

The Allocate node is used to signal that memory should be allocated for a particular tensor. Often inside of a function, Initialize and Allocate nodes are constructed in pairs as part of the AST of a program since, logically, in order to compute a tensor we need to declare its variable and allocate its space. Each function will only initialize and allocate those tensors that it did not receive storage for.

Next, a Statement can also be subclassed as a Definition, which is a node describing a tensor computation, potentially in terms of other tensors. A lot of work has been

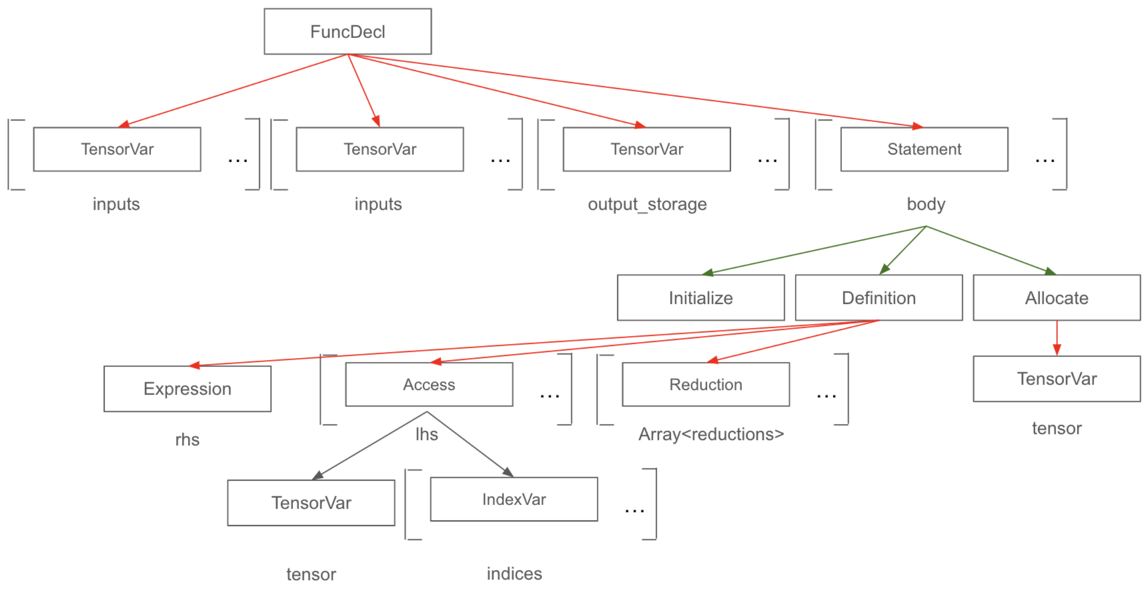


Figure 4-3: Function Declaration

done on optimizing individual tensor definitions like this, so GSTACO will determine when a Definition node can be offloaded to Finch rather than handled internally. Definitions can sometimes have multiple tensor outputs (one multiple-output kernel), so its left-hand-side consists of an array of tensor write accesses instead of just one. The right-hand-side of a Definition inherits from the Expression node. Additionally, we also keep track of an Array of Reduction nodes for each tensor definition. As exemplified in Chapter 3, the extended Einsum notation that we use allows for possibly several reductions, with user-specified operator and initial values. This information is stored inside the Reduction nodes, whose order in the array coincides with the user-specified order and dictates which loops will be reduced first.

4.1.4 Expression

There are 6 subtypes of Expressions in the high level IR of GSTACO (we extend this high level IR with additional nodes that are needed in the low level IR in order to make code generation easier; each will be described in the Lowering phase). The base types correspond to literals (Literal), read access into tensors (ReadAccess),

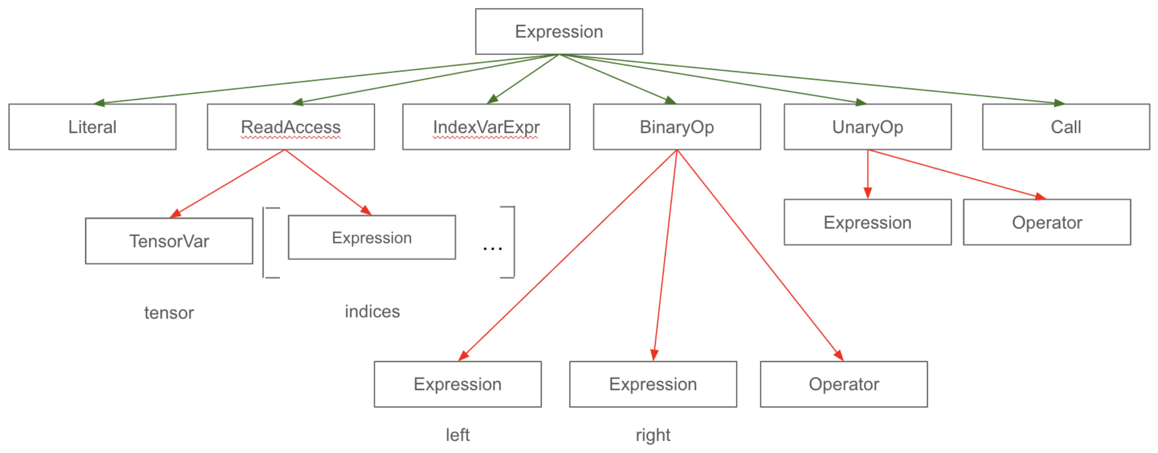


Figure 4-4: Expression subtypes

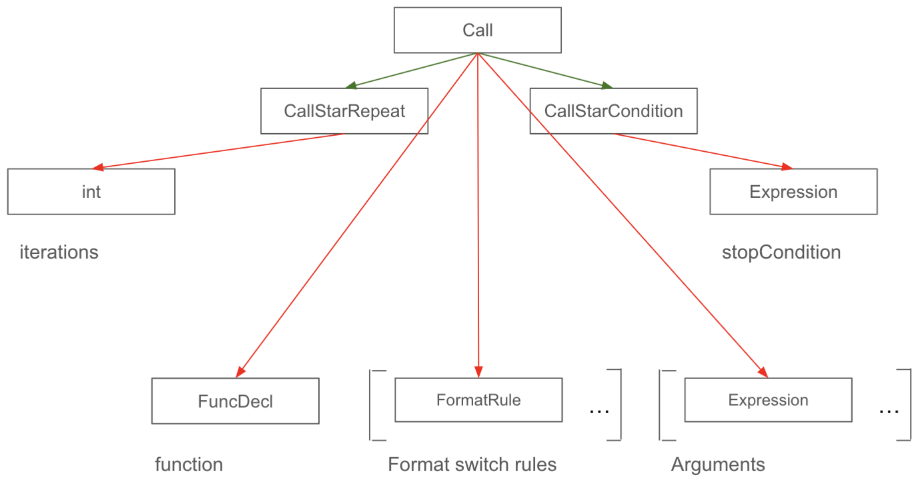


Figure 4-5: Call subtypes

and index variable expressions (IndexVarExpr). The recursive types are for binary expressions (BinaryOp - arithmetic,comparison,boolean), unary operator expressions (UnaryOp - boolean not), and function calls (Call). Additionally, there are two special types of functional calls which inherit from Call. The star operator can be applied to a function call either for a fixed number of iterations (CallStarRepeat) or until a stopping condition is met (CallStarCondition). Each of these contains a set of rules (FormatRule) dictating how and when the formats of certain tensors involved should change throughout iterations. These are illustrated in 4-4 and 4-5.

Lastly, while most nodes in the class inheritance hierarchy are concrete types, some of them are virtual and are not meant to ever be instantiated. For example, ModuleComponent, Statement, and Expressions are types which only exist to help the compiler architect and enhance the Object Oriented Design.

4.2 Parser

The high level IR is the product of the GSTACO parser, which is a generated parser reading in GSTACO syntax and building the program AST using the previously described classes as building blocks. It is created from a context-free version of GSTACO's grammar, using the Flex lexical analyzer and the Bison general-purpose parser generator. Bison takes in a grammar and tokens definition and generates a kind of bottom-up deterministc shift-reduce parser. The grammar and list of tokens we used can be found in the appendix figures A-1 and A-2 respectively.

4.3 IR Cleanup

Another drawback of Bison is that it requires our grammar to be context free. Nevertheless, some AST nodes contain information that is very much dependent on other ascendant nodes. For example, a Call node stores a pointer to its corresponding FuncDecl node (the declaration of the function it calls). This may be a node higher up the hierarchy that was already individually instantiated. Therefore, in order to correctly link it to the call expression, we would need to keep track of some form of state in the parser, remembering which functions have already been instantiated. Seeing as this is not possible in our parser due to Bison limitations, we decided to create call nodes without a link to their functions, and then introduce a separate cleanup step. In this phase, we use a custom IR rewriter class in order to visit the entire AST recursively and augment the call nodes with function pointer information. Since we define the rewriter as a visitor over the IR, we can define and maintain as much state as needed. We applied this same approach to fix other parser-related issues too.

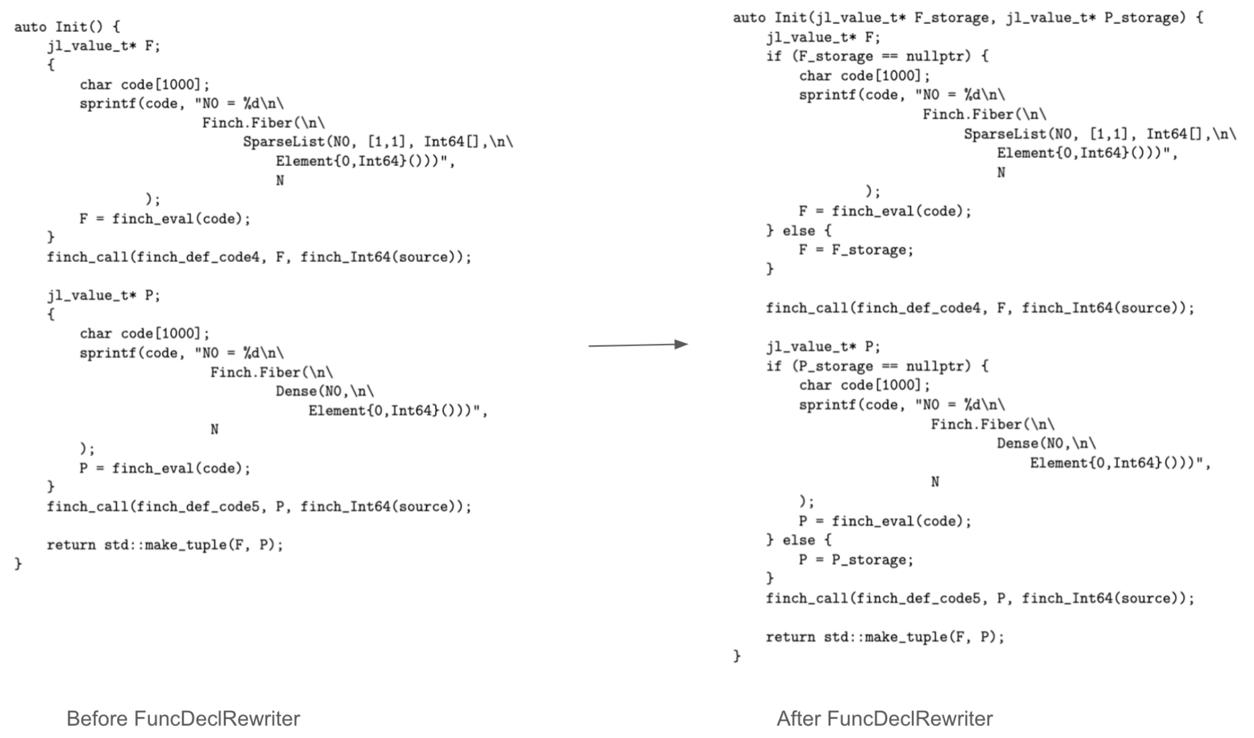
4.4 Lowering

After constructing the IR and cleaning it up, the next step is to generate a low level IR which resembles C++ and Finch code more closely, and hence makes code generation easier.

The first observation we made was that memory allocation may or may not happen for a particular tensor output (see memory reuse section), and switching this feature on or off would be much easier if we could insert/remove a special IR node rather than crowding the code with if statements. Therefore, we augment the IR with the Allocate node, which we add to the AST after the Initialize nodes of tensors that don't reuse memory. We do this using the `AllocateInserter` class.

Next, we had to modify the IR in order to maintain the previously stated invariant: callers allocate memory for outputs and pass it to calees as pointers. Since this is an internal invariant and not obvious at all from the GSTACO frontend, the parser could not have appropriately built the function declarations to take additional arguments for the memory pointers. Similarly, the parser is not responsible for refactoring the Call nodes and add argument nodes corresponding to storage. As such, we added two more IR passes, the `FuncDeclRewriter` and `MemoryReuseRewriter`. Figures 4-6 and 4-7 emphasize the effect of these rewriters on the generated code in the context of some BFS snippets.

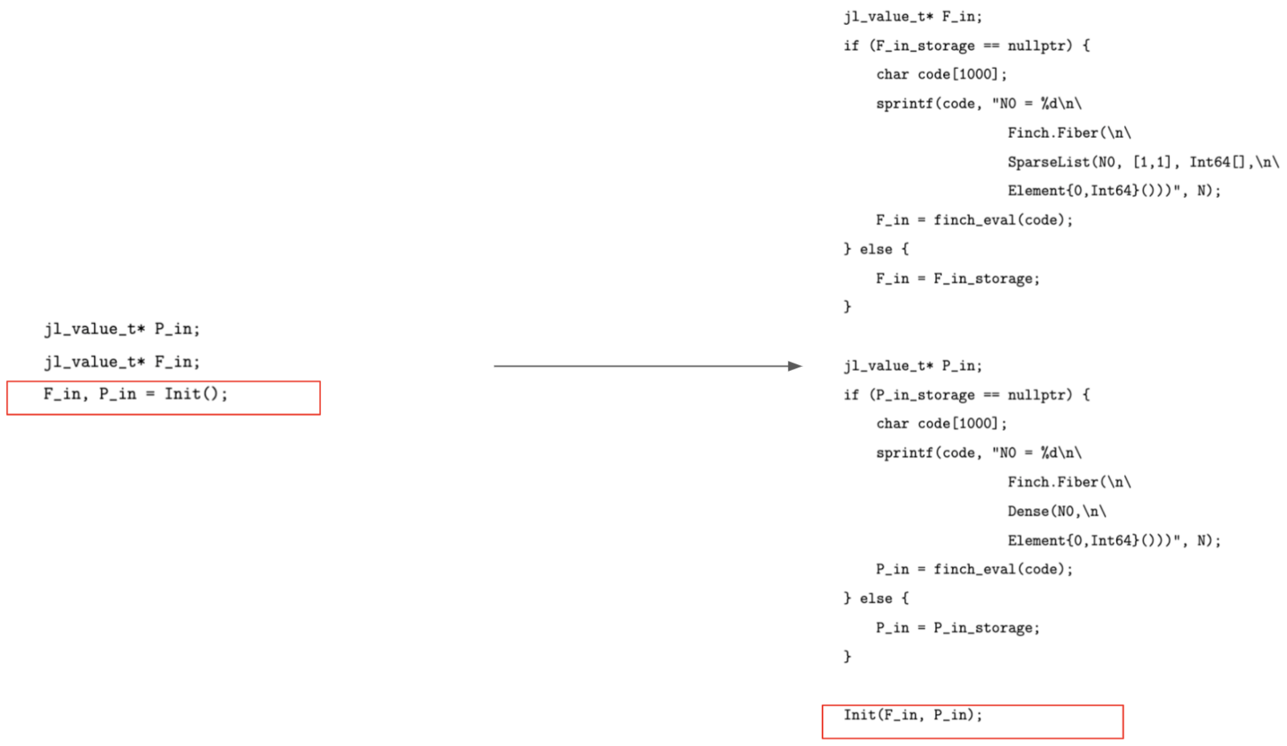
After paving the way for easier memory management, we start thinking about lowering the multiple-output function calls. As seen in GSTACO syntax, it's possible to unpack the return value of a function into multiple variables (even place holders). However, this is not supported in C++, where unpacking would first require storing the result in a tuple variable, and then initializing each output with the corresponding tuple element (using std::get). As such, we introduce two more nodes in our IR. `MultipleOutputDefinition`, `TupleVar`, and `TupleVarReadAccess`. The former describes a definition whose right hand side is a function call with multiple outputs. Its left hand side is a `TupleVar` nodes. This statement would then be followed by definitions which read from the tuple variable and store into each of the desired output



```
auto Init() {
    jl_value_t* F;
    {
        char code[1000];
        sprintf(code, "N0 = %d\n\
                       Finch.Fiber(\n\
                            SparseList(N0, [1,1], Int64[],\n\
                                Element{0,Int64}()))",
                                N
                );
        F = finch_eval(code);
    }
    finch_call(finch_def_code4, F, finch_Int64(source));

    jl_value_t* P;
    {
        char code[1000];
        sprintf(code, "N0 = %d\n\
                        Finch.Fiber(\n\
                                Dense(N0,\n\
                                    Element{0,Int64}()))",
                        N
        );
        P = finch_eval(code);
    }
    finch_call(finch_def_code5, P, finch_Int64(source));

    return std::make_tuple(F, P);
}
```

```
auto Init(jl_value_t* F_storage, jl_value_t* P_storage) {
    jl_value_t* F;
    if (F_storage == nullptr) {
        char code[1000];
        sprintf(code, "N0 = %d\n\
                       Finch.Fiber(\n\
                            SparseList(N0, [1,1], Int64[],\n\
                                Element{0,Int64}()))",
                                N
                );
        F = finch_eval(code);
    } else {
        F = F_storage;
    }

    finch_call(finch_def_code4, F, finch_Int64(source));

    jl_value_t* P;
    if (P_storage == nullptr) {
        char code[1000];
        sprintf(code, "N0 = %d\n\
                        Finch.Fiber(\n\
                                Dense(N0,\n\
                                    Element{0,Int64}()))",
                        N
        );
        P = finch_eval(code);
    } else {
        P = P_storage;
    }
    finch_call(finch_def_code5, P, finch_Int64(source));

    return std::make_tuple(F, P);
}
```

Figure 4-6: FuncDeclRewriter : modify function declarations to take allocated memory as an input pointer, and only allocate memory if the pointer is null



```
jl_value_t* P_in;
jl_value_t* F_in;
F_in, P_in = Init();
```

```
jl_value_t* F_in;
if (F_in_storage == nullptr) {
    char code[1000];
    sprintf(code, "N0 = %d\n\
                Finch.Fiber(\n\
                SparseList(N0, [1,1], Int64[],\n\
                Element{0,Int64}()))", N);
    F_in = finch_eval(code);
} else {
    F_in = F_in_storage;
}

jl_value_t* P_in;
if (P_in_storage == nullptr) {
    char code[1000];
    sprintf(code, "N0 = %d\n\
                Finch.Fiber(\n\
                Dense(N0,\n\
                Element{0,Int64}()))", N);
    P_in = finch_eval(code);
} else {
    P_in = P_in_storage;
}

Init(F_in, P_in);
```

Figure 4-7: MemoryReuseRewriter : refactors function calls to pass pointers to allocated memory from the caller

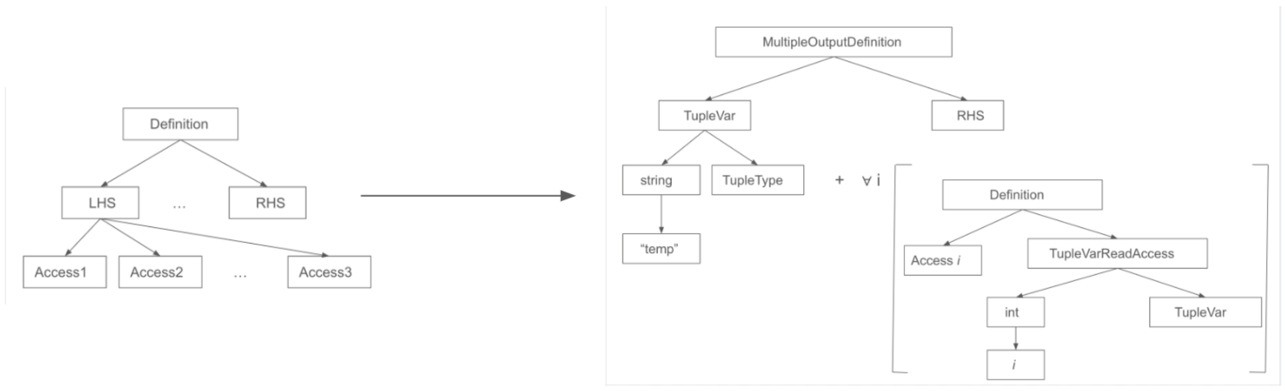
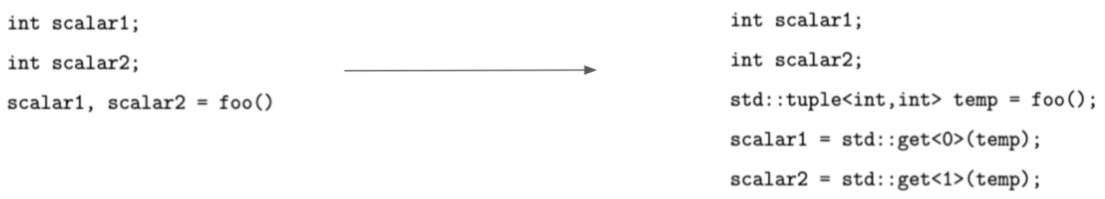


Figure 4-8: CallRewriter effect on IR: split a Definition into a MultipleOutputDefinition (which computes a tuple of results) and a set of Definitions which index into the tuple to compute final results



```
int scalar1;
int scalar2;
scalar1, scalar2 = foo()
```

```
int scalar1;
int scalar2;
std::tuple<int,int> temp = foo();
scalar1 = std::get<0>(temp);
scalar2 = std::get<1>(temp);
```

Figure 4-9: CallRewriter effect on pseudocode

variables. Their right-hand-side is therefore a `TupleVarReadAccess`. Figures 4-8 and 4-9 show how this transformation affects the IR and pseudocode, respectively.

Nevertheless, we still have a construct that doesn't map quite well to C++ code: the outer loops `CallStarRepeat` and `CallStarCondition`. In these cases, we would like to make it more obvious in the low level IR that there is a temporary variable storing the loop breaking condition (`stopCondition`). We therefore need to create a new Definition whose right hand side evaluates to a boolean and whose left hand side introduce a new temporary which should be checked every iteration. When the value becomes true, the loop can be broken out of. We can not simply insert this new Definition node before the call node, as in this case the condition is only evaluated once, right before the loop. Therefore, we add it as a new field to the call node called

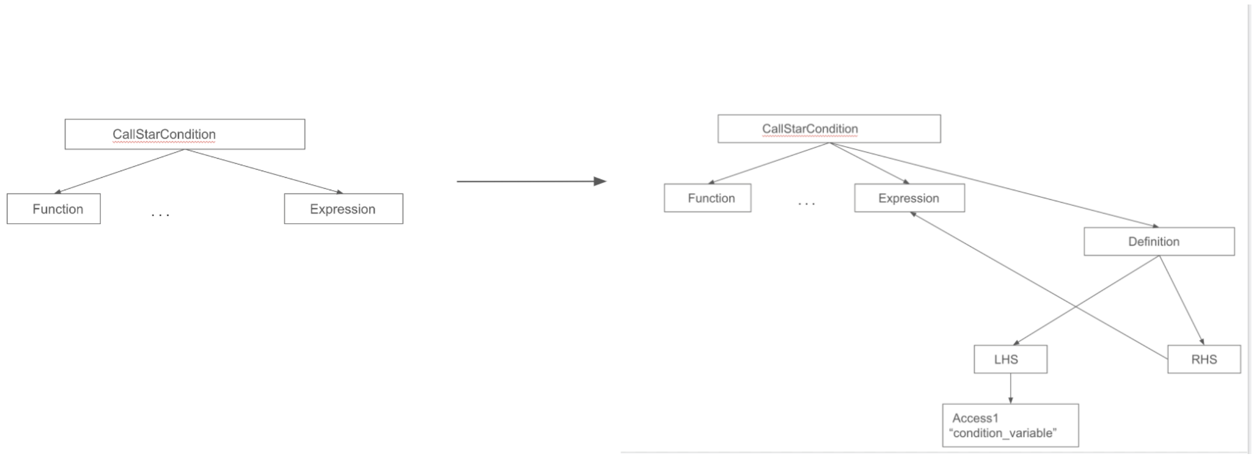


Figure 4-10: CallStarConditionRewriter

`conditionDefinition`. This way, code generation will know it has to emit code for that Defintion inside the generated `while` loop. This transformation is pictured in figure 4-10.

Lastly, after we applied all the rewriters that can potentially generate new definitions, we would like to make another pass and assign unique numerical identifiers to each definition, as needed in code generation. This is achieved with the `DefinitionSplitter` class.

Figure 4-11 shows the relative order in which all these visitors are applied.

4.5 Optimizations

GSTACO performs a range of intra-kernel and cross-kernel optimizations, with the former being offloaded to the Finch tensor compiler and the latter being supported in GSTACO's own pipeline. GSTACO does not aim to reimplement already existing algorithms for single-tensor sparse computation. Rather, it tries to optimize across multiple sparse computations in order to produce full-fledged tensor applications. This has not, to the best of our knowledge, been done before.

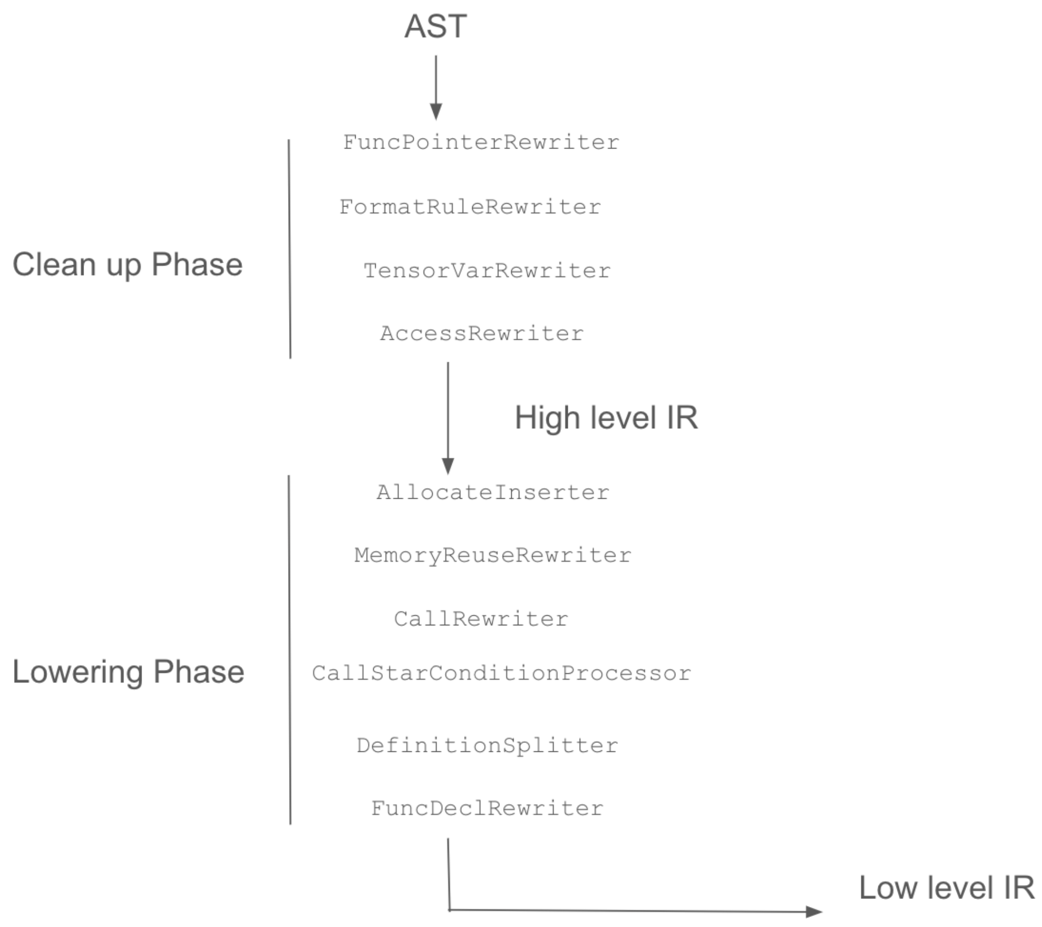


Figure 4-11: Rewriters

4.5.1 Cross-Kernel

A good example that showcases our contribution so far is the memory reuse optimization across functions and outer loops. A naive and initial approach to memory management would be to simply allocate memory whenever a new tensor is computed. This, however, leads to a lot of redundant allocations, thus yielding a sub-optimal outcome in terms of both runtime and storage. For example, both the caller and the callee of a function technically need to produce an output, and so they would both have to allocate memory for the same object. We already touched on the solution for this when we introduced the `MemoryReuseRewriter` and `FuncDeclRewriter`. However, those rewriters don 't completely solve the issue of repeated allocation in the Star operator case.

With the repeated call nodes, while each function call reuses memory from the caller, we also need to have each iteration reuse memory from the previous iteration (with the only allocation happening at the very beginning of the loop). One way to achieve this is to use an intermediary temporary variable in order to swap the contents of the inputs and outputs every iteration such that we never need more than two memory locations for each input-output pair. Therefore, a definition like `"C, D = foo(A, B) | 3"` should yield C++ code resembling the pseudocode below:

```
int iter = 0;
// Copy the inputs so they don’t get modified while swapping
in_A = Copy A; in_B = Copy; B
out_A = Allocate A; out_B = Allocate B;
while(iter < 3) {
    // foo constructs the result directly into out_A and out_B
    foo(in_A, in_B, out_A, out_B);
    temp_A = in_A; temp_B = in_B;
```

```
        in_A = out_A; in_B = out_B;
        out_A = temp_A; out_B = temp_B
        iter += 1;
    }
    C = out_A; D = out_B;
```

This is just one optimization that we implemented, but we are confident that GSTACO's IR and design are expressive enough to support many other optimizations in-between kernel invocations, including data-flow.

4.5.2 Intra-Kernel

Optimizations within tensor expressions stem from opportunities to cut down on computation and storage based on the sparsity of the input. These are best handled by state of the art tensor compilers like TACO and, more recently, Finch.

Sparsity Generator: TACO

Our first approach was centering around TACO, which is not only a DSL but also a C++ library emitting C++ code. This makes it easy to integrate within our compiler with little to no overhead. GSTACO would involve a code generation pass which transforms all kernels into TACO data structures. It would then compile them and capture TACO's output (C++ code), which would then be included in the final source code. This way, we could invoke TACO and integrate its result into our own compilation product.

On the downside, it turned out that TACO had drawbacks that we could not fix without changing its design significantly, which was beyond the scope of our project. For instance, TACO does not efficiently handle computations where the left hand side tensor is sparse. This was a nonnegotiable for our project since many tensor algorithms require sparse outputs (the nodes frontier in BFS for example). Additionally, TACO also did not support kernels with multiple reductions using different operators, or computing multiple outputs within the same loop. Since fixing all of

TACO's insufficiency's would have been an entire project in and of itself, we decided to continue our search for tensor compilers.

Sparsity Generator: Finch

While Finch's design does not suffer from the same issues as TACO's (Finch does support sparse and multiple outputs and also has a very comprehensive interface for reductions), it is less compatible with GSTACO's infrastructure. Finch is written in the Julia programming language and emits Julia code. Because of this, we could not use it directly, neither as a DSL compiler, nor as a library. We had to use Finch's C interface, which passes values around between the C and the Julia runtime in order to integrate the two. For example, a tensor could be represented in Finch as a `Finch.Fiber` object, and the C embedding can convert this into a `jl_value_t*` pointer usable in C/C++. Similarly, one can call julia functions from a C/C++ environment by first evaluating and converting them to a `jl_function_t*` pointer.

This extra step between C and Julia adds not only additional complexity to GSTACO's code generation, but also overhead to its performance. We believed that this overhead, while not insignificant, is secondary to other performance improvements that we decided to focus on first hand. Additionally, other approaches that have a lower impact on performance increase the code complexity of integration even further (see Further Work for an alternative embedding solution).

In the light of the above, we decided to use Finch as underlying tensor compiler and connect it to GSTACO via its C embedding module.

4.6 Code Generation

The code generation phase consists of a series of visitors on the product of the lowering phase and eventually produces 3 files: a C++ source/header pair as well as a driver which calls the main function of the program.

Some of the codegen visitors are used for bookkeeping (helpers) and some actually emit code to be embedded into the source file. Without going into a lot of detail about the helpers, this section will cover how we generate code for `Definition`'s as well as how we setup Finch for automated use.

The first thing we concern ourselves with is finding out which IR constructs can directly emit code and which need some help from Finch. In general, any definition which either computes a tensor from scratch or contains tensor expressions on the right hand side, will be offloaded to Finch. Another case where we invoke Finch is memory allocation (Allocate nodes) of tensor variables (not scalars), as we need to inform the Julia runtime of these new objects. On the other hand, definitions and allocations that only involve scalars as well as all other language constructs (function definitions, function calls, initializations, etc) are directly lowered to C++ code by GSTACO. To perform this analysis, we use the `NeedsFinchVisitor` which produces a mapping between each node and a boolean representing whether it needs Finch or can be handled by GSTACO alone. A few examples of nodes that do and don't need Finch can be found in figure 4-14.

```
A[i][j] = 10
a = A[i]| i:(+, 0)
A[i] = B[i][j] * C[j] | j:(+, 0)
```

Figure 4-12: with Finch

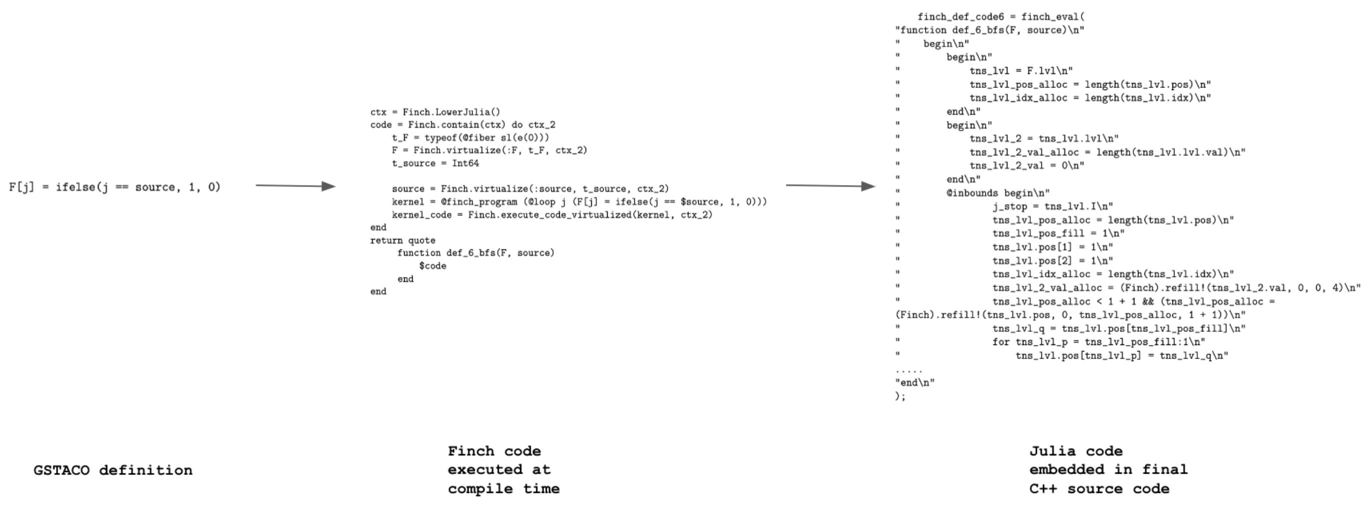
```
c, d = foo(1, 2)
c, d = foo(1, 2) | 5
c = 1
d = c * 3
```

Figure 4-13: without Finch

Figure 4-14: NeedsFinchVisitor

According to this analysis, we will then perform another pass over the IR nodes that do require Finch. In this phase, we build the equivalent Finch kernels out of each GSTACO definition and compile them down to free-standing Julia code. The generated source will contain this Julia implementation wrapped in a call to the Julia-C interface. This will, at runtime, produce the function pointer needed to call the compiled kernel and thus compute the `Definition`. This step is performed by the FinchCompileVisitor and is pictured in 4-15.

While this usage of Finch might seem intricate, we chose this approach in order to separate Finch's compile time from its run time. This is crucial so that the code that



```
F[j] = ifelse(j == source, 1, 0)
```

```
ctx = Finch.LowerJulia()
code = Finch.contain(ctx) do ctx_2
    t_F = typeof(@fiber sl(e(0)))
    F = Finch.virtualize(:F, t_F, ctx_2)
    t_source = Int64

    source = Finch.virtualize(:source, t_source, ctx_2)
    kernel = @finch_program (@loop j (F[j] = ifelse(j == $source, 1, 0)))
    kernel_code = Finch.execute_code_virtualized(kernel, ctx_2)
end
return quote
     function def_6_bfs(F, source)
         $code
     end
end
```

```
    finch_def_code6 = finch_eval(
"function def_6_bfs(F, source)\n"
"    begin\n"
"        begin\n"
"            tns_lvl = F.lvl\n"
"            tns_lvl_pos_alloc = length(tns_lvl.pos)\n"
"            tns_lvl_idx_alloc = length(tns_lvl.idx)\n"
"        end\n"
"        begin\n"
"            tns_lvl_2 = tns_lvl.lvl\n"
"            tns_lvl_2_val_alloc = length(tns_lvl.lvl.val)\n"
"            tns_lvl_2_val = 0\n"
"        end\n"
"        @inbounds begin\n"
"                j_stop = tns_lvl.I\n"
"                tns_lvl_pos_alloc = length(tns_lvl.pos)\n"
"                tns_lvl_pos_fill = 1\n"
"                tns_lvl.pos[1] = 1\n"
"                tns_lvl.pos[2] = 1\n"
"                tns_lvl_idx_alloc = length(tns_lvl.idx)\n"
"                tns_lvl_2_val_alloc = (Finch).refill!(tns_lvl_2.val, 0, 0, 4)\n"
"                tns_lvl_pos_alloc < 1 + 1 && (tns_lvl_pos_alloc =
(Finch).refill!(tns_lvl.pos, 0, tns_lvl_pos_alloc, 1 + 1))\n"
"                tns_lvl_q = tns_lvl.pos[tns_lvl_pos_fill]\n"
"                for tns_lvl_p = tns_lvl_pos_fill:1\n"
"                    tns_lvl.pos[tns_lvl_p] = tns_lvl_q\n"
.....
"end\n"
);
```

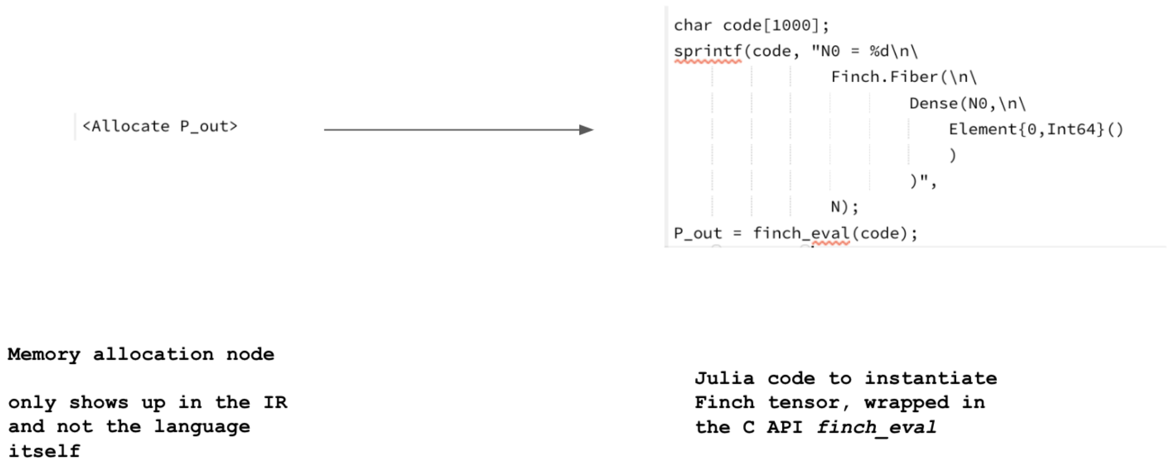
GSTACO definition
Finch code executed at compile time
Julia code embedded in final C++ source code

Figure 4-15: FinchCompileVisitor on Definition node

GSTACO generates does not incur the performance cost of compiling Finch to Julia, a process which tends to be very slow. We will still incur the cost of the Julia compile time (since Julia-C conversion happens at runtime), but that only happens the first time we run the Main function (also called warm up). The Julia environment caches compilation results, so subsequent executions (within the same process) should not suffer from this downside.

Besides Definitions, Allocation nodes are also lowered to julia code which is then embedded in the source code, as exemplified in 4-16.

Other notable constructs which don't map one-to-one with C++ code are the star operator (`CallStarRepeat` and `CallStarCondition`) and format change rules (`FormatRule`). The former are lowered down to `for` and `while` loops respectively, while the latter generates an if statement to check whether the condition for the tensor format switch has been reached.


<Allocate P_out>
Memory allocation node
only shows up in the IR and not the language itself
Julia code to instantiate Finch tensor, wrapped in the C API *finch_eval*

```
char code[1000];
sprintf(code, "N0 = %d\n\
                Finch.Fiber(\n\
                        Dense(N0,\n\
                            Element{0,Int64}()
                            )
                        )",
                N);
P_out = finch_eval(code);
```

Figure 4-16: FinchCompileVisitor on Allocation node

Chapter 5

Evaluation

GSTACO is in its early stages, so for now it is primarily a "new-feature" contribution, rather than a "performance" contribution. We reiterate that GSTACO unlocks new capabilities absent from other tensor compilers. We also analyze its current performance (with little optimization in place), and discuss how our design accommodates future performance contributions which will bring this project closer in speed to older state-of-the-art compilers.

5.1 Capabilities

Unlike TACO and Finch, which only compute individual tensor definitions, GSTACO can encode entire applications with the help of outer loop and scheduling constructs. As detailed in previous chapters, GSTACO can handle cross-kernel tensor format changes and data flow. This means that the user does not need to worry about bridging the gap between the different expressions that constitute an algorithm. In the most basic case, this gap might just include passing data around from one tensor kernel to another. In other scenarios, it may require more complex control flow like `if/for/while` blocks and memory management. The GSTACO language is powerful enough to fill in these gaps without any additional effort on the programmer's side. We prove this by testing that GSTACO correctly generates code for 5 different tensor-based graph algorithms. Implementations for PageRank, Breadth First Search, Single

Source Shortest Paths, Betweenness Centrality, as well as Connected Components can be found in the attached appendix. We also show that the IR preserves opportunities for optimization by implementing memory reuse across the outer loops, as described before. We believe this should be suggestive of GSTACO's ability to be further optimized.

5.2 Performance

While the main focus of this project was to produce a compiler with new abilities which works well on a variety of inputs, the ultimate hope is for GSTACO to also be a high performance compiler, producing output comparable to heavily optimized manual implementations. All design choices have been made with this target in mind, taking into account future performance extensions. Therefore, we provide some initial benchmarks to show where GSTACO stands now and to serve as reference for further work.

5.2.1 Experimental Setup

We compiled PageRank with both GSTACO and Graphit and measured the performance of the generated code on different graph datasets. Since Graphit has already seen comprehensive work done on optimizations and scheduling, we turn off the optimizations that are not also present in GSTACO in order to level the ground and get a better idea about how GSTACO performs relative to its age.

The largest graph used was the LiveJournal dataset from the Stanford Network Analysis Platform (SNAP) [17] for C++, which has approx. 5 million nodes and 86 million edges. In order to test on smaller sized graphs as well, we used subgraphs of the SNAP dataset, which we computed by first finding a minimum spanning tree containing the desired number of nodes. We then started randomly adding edges form the large graph whose ends were present in the established tree, taking care to maintain the initial node-to-edge ratio. All benchmarks have been run on an AMD processor, model EPYC 7642 48-Core, architecture x86-64, frequency 2.3GHz, and the following cache configuration: L1d 512KiB, L1i 512KiB, L2 4MiB, L3 16MiB.

5.2.2 Results

With the above setup, GSTACO-compiled PageRank runs from less than a second on graphs of 250000 nodes to around 22 seconds on a sparse graph of 5 million nodes. Results for this algorithm on various graph sizes can be found in figure 5-1.

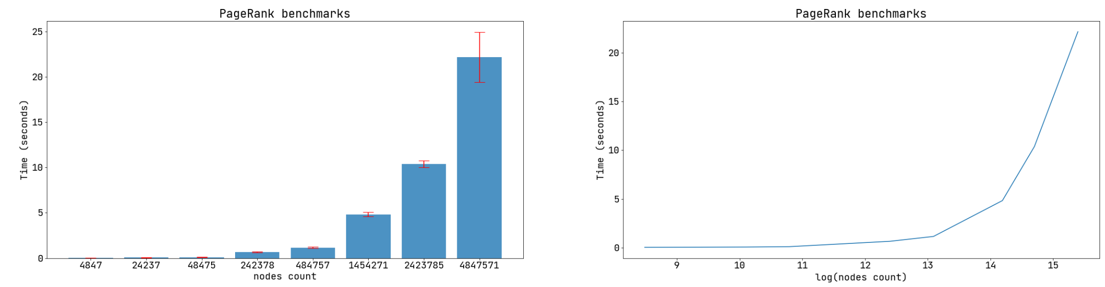


Figure 5-1: GSTACO execution times vs. number of nodes in the graph for PageRank

When we run PageRank generated by Graphit without additional optimizations (like parallelization), we get worse perfromance, as seen in figure 5-2. However, with optimizations and proper schedules, Graphit can run pagerank on large graphs in less than 10 seconds, a number we should also aim for in the future development of GSTACO.

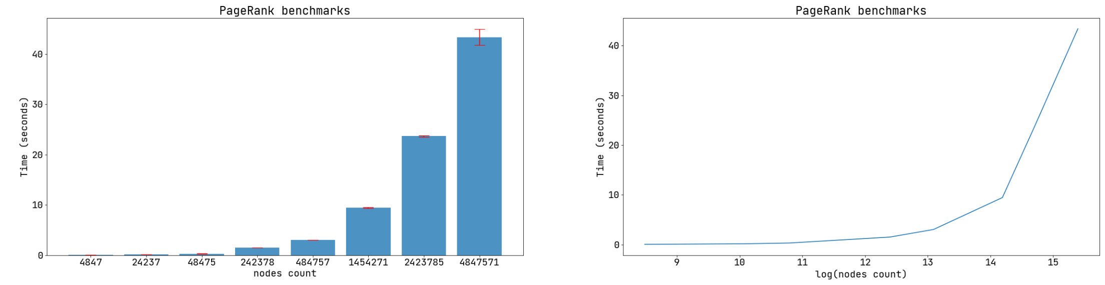


Figure 5-2: Graphit benchmark on PageRank with optimizations turned off

Chapter 6

Further Work

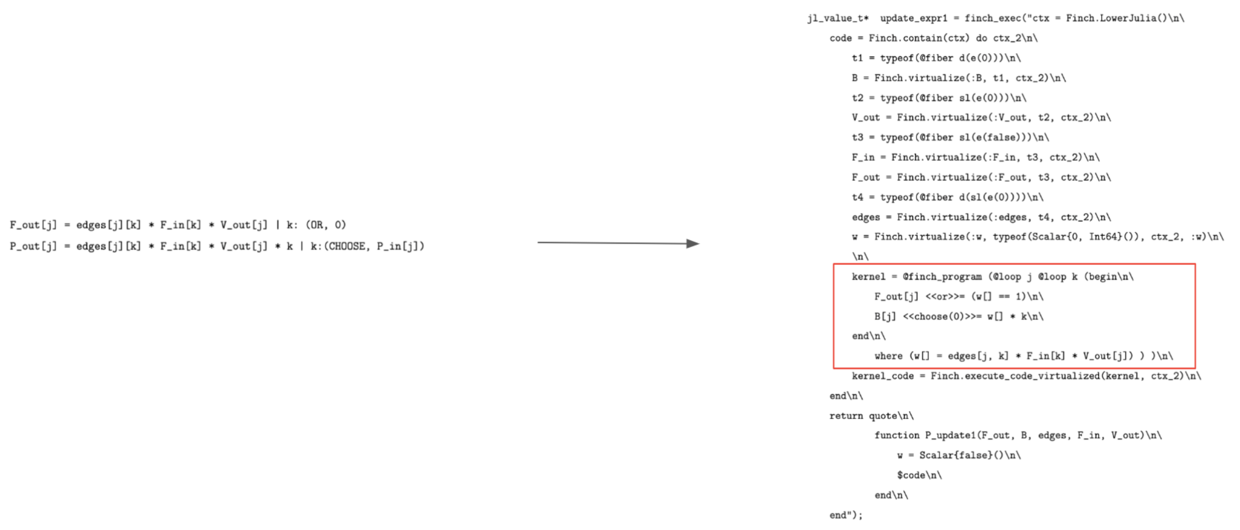
The presented work serves as a proof of concept for the feasibility of GSTACO as an improved tensor algebra language. While we believe the evidence should be convincing, it has to be noted that there is still a lot of work needed to bring GSTACO closer to the state-of-the art by industry standards.

6.1 New Applications

The lowest hanging fruit would be adding more applications to GSTACO's repertoire. We believe GSTACO is suitable for tensor-expressible programs that Graphit or GraphBLAS don't support, like hypergraph analysis [8] (graphs where edges are many-to-many node mappings), solvers (Sudoku), or graph neural networks [19] (GNN). Making sure GSTACO generates efficient code in these new algorithm spheres will bring us closer to our generality goal.

6.2 Performance Tuning

As mentioned in the previous chapter, GSTACO could also benefit from additional performance optimizations which can be achieved through a combination of more advanced IR visitors and scheduling constructs. In other words, these are optimizations that stem from the compiler's own analysis and optimizations that are driven by the



```
F_out[j] = edges[j][k] * F_in[k] * V_out[j] | k: (OR, 0)
P_out[j] = edges[j][k] * F_in[k] * V_out[j] * k | k:(CHOOSE, P_in[j])
```

```
jl_value_t*  update_expr1 = finch_exec("ctx = Finch.LowerJulia()\n\
    code = Finch.contain(ctx) do ctx_2\n\
        t1 = typeof(@fiber d(e(0)))\n\
        B = Finch.virtualize(:B, t1, ctx_2)\n\
        t2 = typeof(@fiber sl(e(0)))\n\
        V_out = Finch.virtualize(:V_out, t2, ctx_2)\n\
        t3 = typeof(@fiber sl(e(false)))\n\
        F_in = Finch.virtualize(:F_in, t3, ctx_2)\n\
        F_out = Finch.virtualize(:F_out, t3, ctx_2)\n\
        t4 = typeof(@fiber d(sl(e(0))))\n\
        edges = Finch.virtualize(:edges, t4, ctx_2)\n\
        w = Finch.virtualize(:w, typeof(Scalar{0, Int64}()), ctx_2, :w)\n\
        \n\
        kernel = @finch_program (@loop j @loop k (begin\n\
            F_out[j] <<or>>= (w[] == 1)\n\
            B[j] <<choose(0)>>= w[] * k\n\
        end\n\
            where (w[] = edges[j, k] * F_in[k] * V_out[j]) ) )\n\
        kernel_code = Finch.execute_code_virtualized(kernel, ctx_2)\n\
    end\n\
    return quote\n\
            function P_update1(F_out, B, edges, F_in, V_out)\n\
                w = Scalar{false}()\n\
                $code\n\
            end\n\
    end");
```

Figure 6-1: CSE optimization in BFS: both parents (P) and frontier (F) are computed within the same multiple-output Finch kernel

user's scheduling directives.

6.2.1 Additional Optimizations

In terms of analysis-based optimizations that are performed from the inside-out, we emphasize those that target more efficient program structure. As mentioned before, GSTACO needs a way to detect which tensor expressions can reuse computation and merge them into a multiple-output kernel. This optimization is also called loop fusion, and can be achieved by performing a data analysis similar to the Common Subexpression Elimination (CSE) in classic compiler theory. Clasically, the common expression is computed once and stored in a separate temporary variable in order to be reused. A similar process would take place in GSTACO, except that the expressions are considered in the context of loops/kernels. Figure 6.2.1 shows a CSE opportunity in BFS.

6.2.2 Scheduling

Currently, GSTACO has a minimalist scheduling language comprised of the `Ord` and `FormatRule` described earlier. However, there are other parameters that users can tune in order to achieve higher performance, and GSTACO would need to support directives for each of them. More advanced settings could allow the programmer to manage loop paralellization (possibly the most impactful of the remaining optimizations), blocking, and cache accesses. Additionally, Graphit also contains scheduling features for specifying edge traversal direction (Push vs Pull schedules) [22]. The challenge here would be to come up with an abstraction that supports a similar optimization, but which does not lose generality and can therefore be applied to non-graph programs too.

6.3 New Backends

GSTACO's syntax and intermediary representaion have been designed with flexibility in mind. We wanted the compiler to support easier code generation for several backends, including GPU and Swarm. However, at the moment GSTACO only has 1 backend, which generates C++ code, which is then compiled down to traditional CPU assembly. Considering that many tensor-based applications are also data-intensive (very large graphs or neural networks) and hence need to run on specialized hardware, targeting new architectures within GSTACO would be a worthwhile engineering effort.

6.4 Improved Integration

Lastly, further work on the integration between GSTACO and Finch can also bring performance improvements. Because Finch emits Julia, we currently achieve this integration through a layer of indirection: the Julia-C interface. A better approach would be compiling the generated Julia down to LLVM [15] and inlining that into GSTACO's other products of compilation after passing them through an LLVM-compatible compiler like Clang [16]. Another clean-slate approach would be implementing a C/C++ back-end for Finch, which can be done either manually or using the Buildit framework for creating new Domain Specific Languages [4].

The contributions proposed in this chapter may require new features to either GSTACO, Finch, or both.

Chapter 7

Conclusion

In light of the work presented above, we conclude that the Generalized Sparse Tensor Algebra Compiler (GSTACO) is a comprehensive and convenient solution for sparse tensor algebra compilation as well as a successful step towards a high-performance tool. It can translate extensive graph logic, overcoming the capability shortcoming of TACO and Finch, while still maintaining per-kernel performance since it is building on top of Finch.

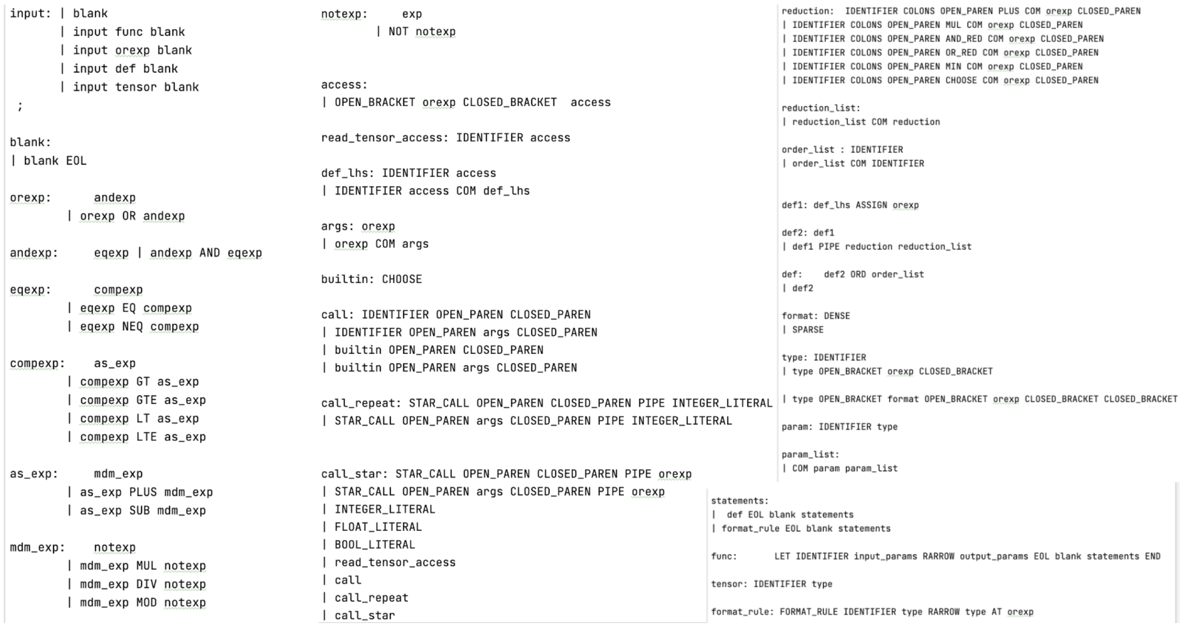
Most importantly, GSTACO's novelty stems from its ability to encode cross-kernel tensor transformations and open the door to a wide range of outer loop optimizations, of which memory reuse has already been implemented. Unlike TACO and Finch, which compute an individual tensor expression per run, GSTACO provides the infrastructure for specifying how multiple tensor computations interact with one another to create full fledged tensor applications. At a lower level, GSTACO also provides a starting scheduling language for establishing loop iteration order, and can be extended to include more complex settings, as dictated by performance requirements. Other features that facilitate the translation of graph algorithms include several reduction operators and statements producing multiple outputs.

We measured GSTACO's performance for PageRank and found that it runs twice as fast as unoptimized Graphit, but much slower than optimized Graphit. We conclude that it still needs more tuning in order to achieve the speed of Graphit's application-specific optimizations while still maintaining its generality. As such, we propose a list of future improvements leveraging the current design, like implementing loop parallelization, commons subexpression elimination, and more complex scheduling directives.

All work can be found on the public GitHub at GSTACO.

Appendix A

Grammar & Token List



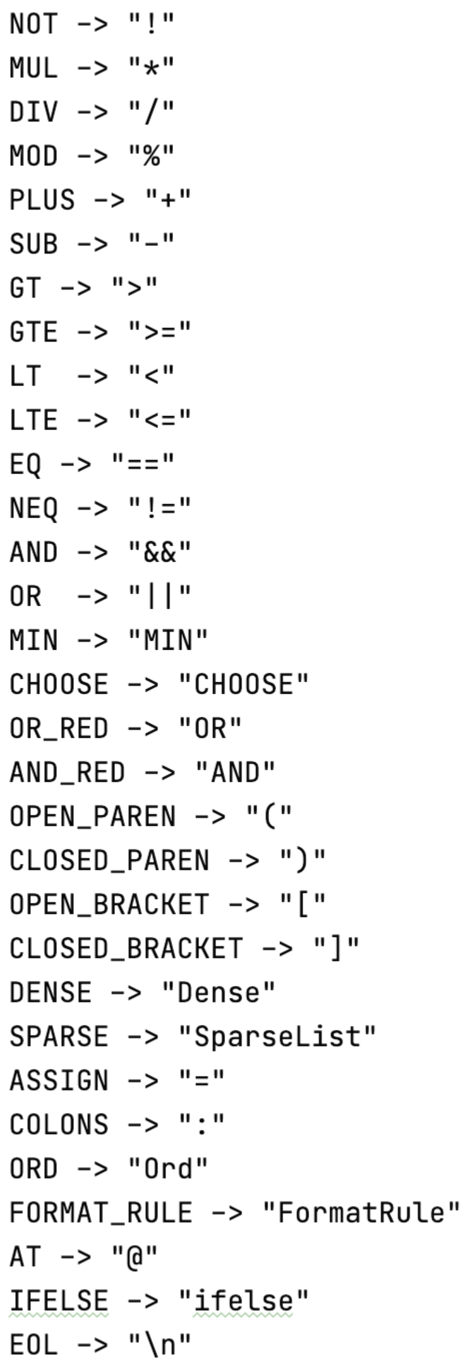
```
input: | blank
       | input func blank
       | input orexp blank
       | input def blank
       | input tensor blank
 ;

blank:
| blank EOL

orexp:      andexp
        | orexp OR andexp

andexp:     eqexp | andexp AND eqexp

eqexp:      compexp
        | eqexp EQ compexp
        | eqexp NEQ compexp

compexp:    as_exp
        | compexp GT as_exp
        | compexp GTE as_exp
        | compexp LT as_exp
        | compexp LTE as_exp


as_exp:     mdm_exp
        | as_exp PLUS mdm_exp
        | as_exp SUB mdm_exp

mdm_exp:    notexp
        | mdm_exp MUL notexp
        | mdm_exp DIV notexp
        | mdm_exp MOD notexp
```

```
notexp:     exp
        | NOT notexp


access:
| OPEN_BRACKET orexp CLOSED_BRACKET  access

read_tensor_access: IDENTIFIER access

def_lhs: IDENTIFIER access
| IDENTIFIER access COM def_lhs

args: orexp
| orexp COM args

builtin: CHOOSE

call: IDENTIFIER OPEN_PAREN CLOSED_PAREN
| IDENTIFIER OPEN_PAREN args CLOSED_PAREN
| builtin OPEN_PAREN CLOSED_PAREN
| builtin OPEN_PAREN args CLOSED_PAREN

call_repeat: STAR_CALL OPEN_PAREN CLOSED_PAREN PIPE INTEGER_LITERAL
| STAR_CALL OPEN_PAREN args CLOSED_PAREN PIPE INTEGER_LITERAL


call_star: STAR_CALL OPEN_PAREN CLOSED_PAREN PIPE orexp
| STAR_CALL OPEN_PAREN args CLOSED_PAREN PIPE orexp
| INTEGER_LITERAL
| FLOAT_LITERAL
| BOOL_LITERAL
| read_tensor_access
| call
| call_repeat
| call_star
```

```
reduction:  IDENTIFIER COLONS OPEN_PAREN PLUS COM orexp CLOSED_PAREN
| IDENTIFIER COLONS OPEN_PAREN MUL COM orexp CLOSED_PAREN
| IDENTIFIER COLONS OPEN_PAREN AND_RED COM orexp CLOSED_PAREN
| IDENTIFIER COLONS OPEN_PAREN OR_RED COM orexp CLOSED_PAREN
| IDENTIFIER COLONS OPEN_PAREN MIN COM orexp CLOSED_PAREN
| IDENTIFIER COLONS OPEN_PAREN CHOOSE COM orexp CLOSED_PAREN

reduction_list:
| reduction_list COM reduction

order_list : IDENTIFIER
| order_list COM IDENTIFIER


def1: def_lhs ASSIGN orexp

def2: def1
| def1 PIPE reduction reduction_list

def:    def2 ORD order_list
| def2

format: DENSE
| SPARSE

type: IDENTIFIER
| type OPEN_BRACKET orexp CLOSED_BRACKET

| type OPEN_BRACKET format OPEN_BRACKET orexp CLOSED_BRACKET CLOSED_BRACKET

param: IDENTIFIER type

param_list:
| COM param param_list
```

```
statements:
|  def EOL blank statements
| format_rule EOL blank statements

func:       LET IDENTIFIER input_params RARROW output_params EOL blank statements END

tensor: IDENTIFIER type

format_rule: FORMAT_RULE IDENTIFIER type RARROW type AT orexp
```

Figure A-1: GSTACO Grammar



```
NOT -> "!"
MUL -> "*"
DIV -> "/"
MOD -> "%"
PLUS -> "+"
SUB -> "-"
GT -> ">"
GTE -> ">="
LT  -> "<"
LTE -> "<="
EQ -> "=="
NEQ -> "!="
AND -> "&&"
OR  -> "||"
MIN -> "MIN"
CHOOSE -> "CHOOSE"
OR_RED -> "OR"
AND_RED -> "AND"
OPEN_PAREN -> "("
CLOSED_PAREN -> ")"
OPEN_BRACKET -> "["
CLOSED_BRACKET -> "]"
DENSE -> "Dense"
SPARSE -> "SparseList"
ASSIGN -> "="
COLONS -> ":"
ORD -> "Ord"
FORMAT_RULE -> "FormatRule"
AT -> "@"
IFELSE -> "ifelse"
EOL -> "\n"
```

Figure A-2: GSTACO Tokens

Appendix B

GSTACO Full Implementations

```
N int
edges int[N][SparseList[N]]

damp float

beta_score float

Let InitRank() -> (r_out float[N])
    r_out[j] = 1.0 / N
End


Let PageRankStep(out_d_in int[N], contrib_in float[N], rank_in float[N],
                r_in float[N]) -> (out_d int[N], contrib float[N], rank float[N],
                r_out float[N])

    out_d[j] = edges[i][j] | i:(+, 0)
    contrib[i] = r_in[i] / out_d[i]
    rank[i] = edges[i][j] * contrib[j] | j:(+, 0.0)
    r_out[i] = beta_score + damp * (rank[i])
End

Let Main() -> (out_d int[N], contrib float[N], rank float[N], r_out float[N])
    out_d[i] = 0
    contrib[i] = 0.0
    rank[i] = 0.0
    _, _, _, r_out = PageRankStep*(out_d, contrib, rank, InitRank()) | 20
End
```

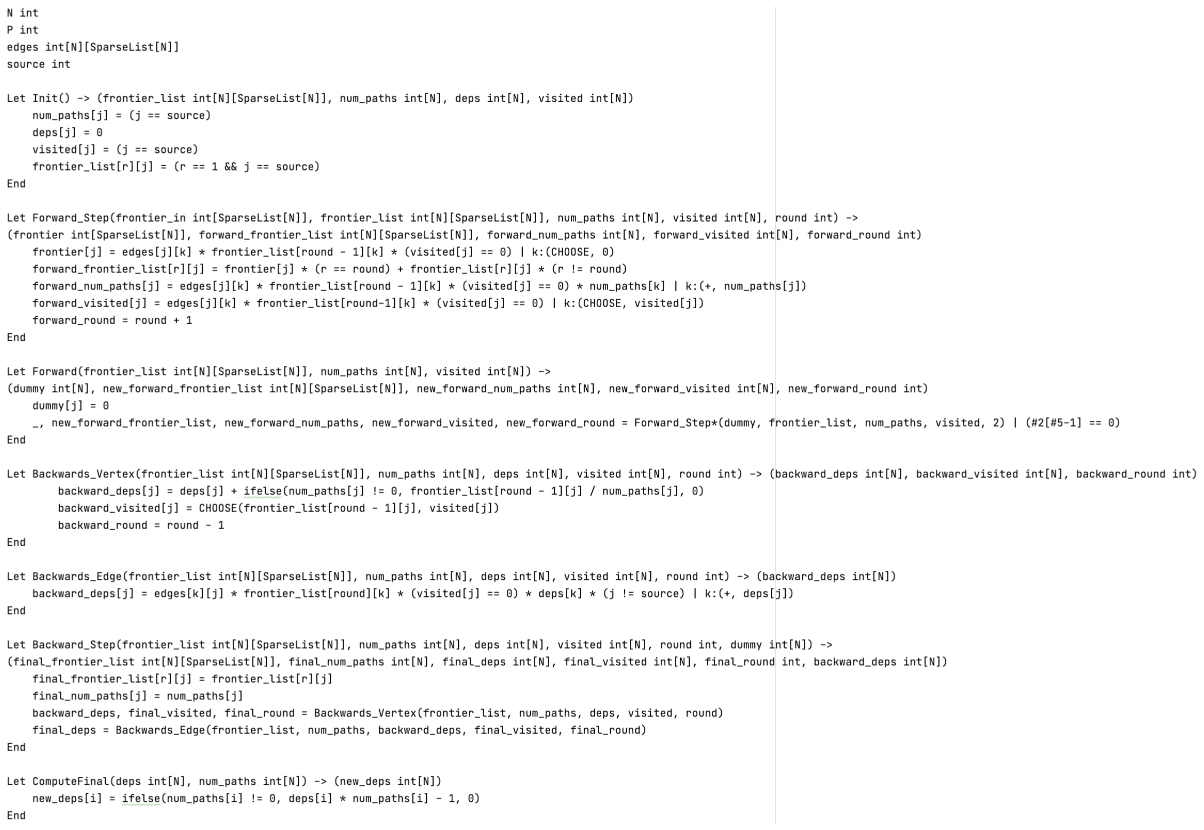
Figure B-1: Page Rank

```
N int

source int

edges int[Dense[N]][SparseList[N]]

Let BFS_Step(F_in int[SparseList[N]], P_in int[Dense[N]], V_in int[N],
        temp_in int[N][1]) ->
        (F_out int[SparseList[N]], P_out int[N], V_out int[N], temp_out int[N][1])

    V_out[j] = P_in[j] == 0 - 1
    F_out[j] = edges[j][k] * F_in[k] * V_out[j] | k: (CHOOSE, 0)
    temp_out[j][m] = edges[j][k] * F_in[k] * V_out[j] * k | k:(CHOOSE, 0)
    P_out[j] = CHOOSE(temp_out[j][1], P_in[j])
End

Let Init() -> (F int[SparseList[N]], P int[N])
    F[j] = (j == source)
    P[j] = (j == source) * (0 - 2) + (j != source) * (0 - 1)
End

Let Main() -> (P_out int[N], F_in int[SparseList[N]], P_in int[N], V int[N], temp_in
    F_in, P_in = Init()
    V[i] = 0
    temp_in[i][j] = 0
    _, P_out, _, _ = BFS_Step*(F_in, P_in, V, temp_in) | (#1 == 0)
End
```

Figure B-2: Breadth First Search

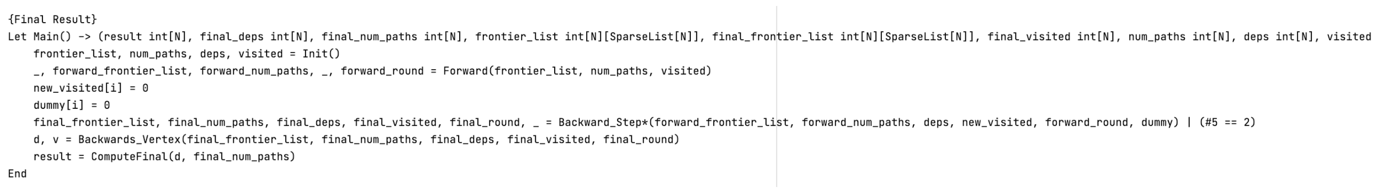
```
N int
P int
edges int[Dense[N]][SparseList[N]]
weights float[Dense[N]][SparseList[N]]
source int

Let Init(source int) -> (dist float[N], priorityQ int[P][SparseList[N]])
    dist[j] = ifelse(j != source, P, 0)
    priorityQ[p][j] = ifelse( (p == 1 && j == source) || (p == P && j != source), 1, 0)
End

Let UpdateEdges(dist float[N], priorityQ int[P][SparseList[N]], priority int, temp_in int[N]) ->
(new_dist float[N], new_priorityQ int[P][SparseList[N]], new_priority int, temp_out int[N])
    temp_out[j] = ifelse(edges[j][k] * priorityQ[priority][k] == 1, weights[j][k] + dist[k], P) | k:(MIN, 2000000000)
    new_dist[j] = MIN(temp_out[j], dist[j])
    new_priorityQ[j][k] = ifelse(
        dist[k] > new_dist[k] && j <= new_dist[k] &&  new_dist[k] < j + 1, 1, ifelse(
                    dist[k] == new_dist[k] && j != priority, priorityQ[j][k], 0
                    )
    )
    new_priority = priority
End

Let SSSP_one_priority_lvl(dist float[N], priorityQ int[P][SparseList[N]], priority int, temp_in int[N]) ->
(new_dist float[N], new_priorityQ int[P][SparseList[N]], new_priority int, temp_out int[N])
    new_dist, new_priorityQ, _, temp_out = UpdateEdges*(dist, priorityQ, priority, temp_in) | (#2[#3] == 0)
    new_priority = priority + 1
End

Let Main() -> (new_dist float[N], dist float[N], priorityQ int[P][SparseList[N]], temp int[N])
    dist, priorityQ = Init(source)
    temp[i] = 0
    new_dist, _, _ = SSSP_one_priority_lvl*(dist, priorityQ, 1, temp) | (#2 == 0 || #3 == (P+1))
End
```

Figure B-3: Single Source Shortest Paths



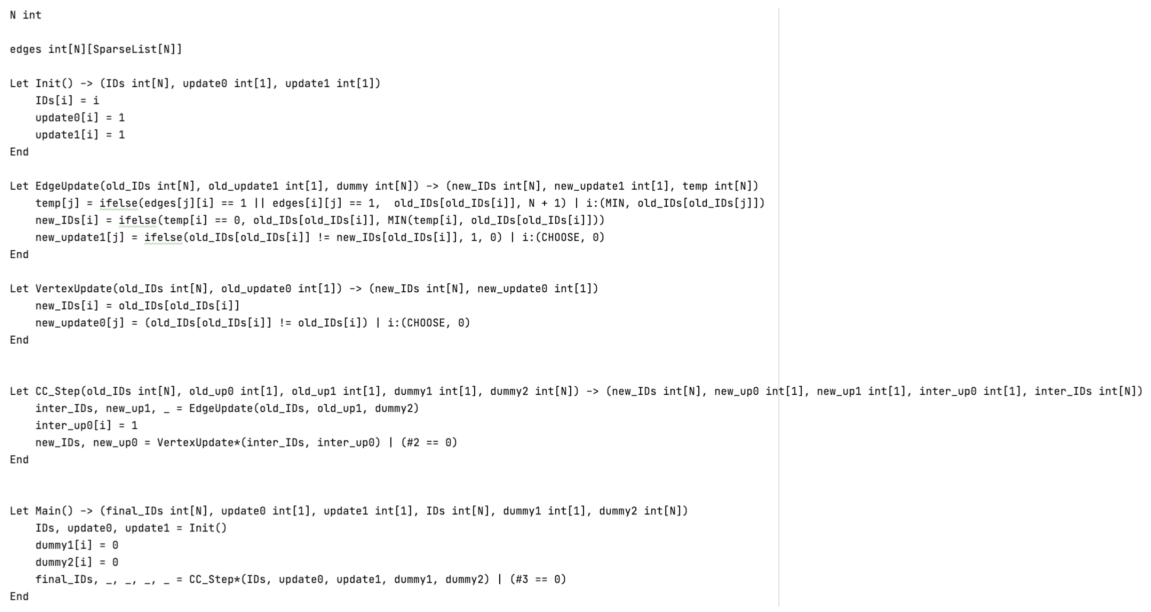
```
N int
P int
edges int[N][SparseList[N]]
source int

Let Init() -> (frontier_list int[N][SparseList[N]], num_paths int[N], deps int[N], visited int[N])
    num_paths[j] = (j == source)
    deps[j] = 0
    visited[j] = (j == source)
    frontier_list[r][j] = (r == 1 && j == source)
End

Let Forward_Step(frontier_in int[SparseList[N]], frontier_list int[N][SparseList[N]], num_paths int[N], visited int[N], round int) ->
(frontier int[SparseList[N]], forward_frontier_list int[N][SparseList[N]], forward_num_paths int[N], forward_visited int[N], forward_round int)
    frontier[j] = edges[j][k] * frontier_list[round - 1][k] * (visited[j] == 0) | k:(CHOOSE, 0)
    forward_frontier_list[r][j] = frontier[j] * (r == round) + frontier_list[r][j] * (r != round)
    forward_num_paths[j] = edges[j][k] * frontier_list[round - 1][k] * (visited[j] == 0) * num_paths[k] | k:(+, num_paths[j])
    forward_visited[j] = edges[j][k] * frontier_list[round-1][k] * (visited[j] == 0) | k:(CHOOSE, visited[j])
    forward_round = round + 1
End

Let Forward(frontier_list int[N][SparseList[N]], num_paths int[N], visited int[N]) ->
(dummy int[N], new_forward_frontier_list int[N][SparseList[N]], new_forward_num_paths int[N], new_forward_visited int[N], new_forward_round int)
    dummy[j] = 0
    _, new_forward_frontier_list, new_forward_num_paths, new_forward_visited, new_forward_round = Forward_Step*(dummy, frontier_list, num_paths, visited, 2) | (#2[#5-1] == 0)
End

Let Backwards_Vertex(frontier_list int[N][SparseList[N]], num_paths int[N], deps int[N], visited int[N], round int) -> (backward_deps int[N], backward_visited int[N], backward_round int)
        backward_deps[j] = deps[j] + ifelse(num_paths[j] != 0, frontier_list[round - 1][j] / num_paths[j], 0)
        backward_visited[j] = CHOOSE(frontier_list[round - 1][j], visited[j])
        backward_round = round - 1
End

Let Backwards_Edge(frontier_list int[N][SparseList[N]], num_paths int[N], deps int[N], visited int[N], round int) -> (backward_deps int[N])
    backward_deps[j] = edges[k][j] * frontier_list[round][k] * (visited[j] == 0) * deps[k] * (j != source) | k:(+, deps[j])
End

Let Backward_Step(frontier_list int[N][SparseList[N]], num_paths int[N], deps int[N], visited int[N], round int, dummy int[N]) ->
(final_frontier_list int[N][SparseList[N]], final_num_paths int[N], final_deps int[N], final_visited int[N], final_round int, backward_deps int[N])
    final_frontier_list[r][j] = frontier_list[r][j]
    final_num_paths[j] = num_paths[j]
    backward_deps, final_visited, final_round = Backwards_Vertex(frontier_list, num_paths, deps, visited, round)
    final_deps = Backwards_Edge(frontier_list, num_paths, backward_deps, final_visited, final_round)
End

Let ComputeFinal(deps int[N], num_paths int[N]) -> (new_deps int[N])
    new_deps[i] = ifelse(num_paths[i] != 0, deps[i] * num_paths[i] - 1, 0)
End
```

Figure B-4: Betweenness Centrality



```
{Final Result}
Let Main() -> (result int[N], final_deps int[N], final_num_paths int[N], frontier_list int[N][SparseList[N]], final_frontier_list int[N][SparseList[N]], final_visited int[N], num_paths int[N], deps int[N], visited
    frontier_list, num_paths, deps, visited = Init()
    _, forward_frontier_list, forward_num_paths, _, forward_round = Forward(frontier_list, num_paths, visited)
    new_visited[i] = 0
    dummy[i] = 0
    final_frontier_list, final_num_paths, final_deps, final_visited, final_round, _ = Backward_Step*(forward_frontier_list, forward_num_paths, deps, new_visited, forward_round, dummy) | (#5 == 2)
    d, v = Backwards_Vertex(final_frontier_list, final_num_paths, final_deps, final_visited, final_round)
    result = ComputeFinal(d, final_num_paths)
End
```

Figure B-5: Betweenness Centrality Main



```
N int

edges int[N][SparseList[N]]

Let Init() -> (IDs int[N], update0 int[1], update1 int[1])
    IDs[i] = i
    update0[i] = 1
    update1[i] = 1
End

Let EdgeUpdate(old_IDs int[N], old_update1 int[1], dummy int[N]) -> (new_IDs int[N], new_update1 int[1], temp int[N])
    temp[j] = ifelse(edges[j][i] == 1 || edges[i][j] == 1,  old_IDs[old_IDs[i]], N + 1) | i:(MIN, old_IDs[old_IDs[j]])
    new_IDs[i] = ifelse(temp[i] == 0, old_IDs[old_IDs[i]], MIN(temp[i], old_IDs[old_IDs[i]]))
    new_update1[j] = ifelse(old_IDs[old_IDs[i]] != new_IDs[old_IDs[i]], 1, 0) | i:(CHOOSE, 0)
End

Let VertexUpdate(old_IDs int[N], old_update0 int[1]) -> (new_IDs int[N], new_update0 int[1])
    new_IDs[i] = old_IDs[old_IDs[i]]
    new_update0[j] = (old_IDs[old_IDs[i]] != old_IDs[i]) | i:(CHOOSE, 0)
End


Let CC_Step(old_IDs int[N], old_up0 int[1], old_up1 int[1], dummy1 int[1], dummy2 int[N]) -> (new_IDs int[N], new_up0 int[1], new_up1 int[1], inter_up0 int[1], inter_IDs int[N])
    inter_IDs, new_up1, _ = EdgeUpdate(old_IDs, old_up1, dummy2)
    inter_up0[i] = 1
    new_IDs, new_up0 = VertexUpdate*(inter_IDs, inter_up0) | (#2 == 0)
End


Let Main() -> (final_IDs int[N], update0 int[1], update1 int[1], IDs int[N], dummy1 int[1], dummy2 int[N])
    IDs, update0, update1 = Init()
    dummy1[i] = 0
    dummy2[i] = 0
    final_IDs, _, _, _, _ = CC_Step*(IDs, update0, update1, dummy1, dummy2) | (#3 == 0)
End
```

Figure B-6: Connected Components

Bibliography

[1] Graphblas. `https://github.com/GraphBLAS`, April 2019. [Online; accessed 7-June-2020].

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[3] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A language for structured coiteration, 2022.

[4] Ajay Brahmakshatriya and Saman Amarasinghe. Buildit: A type-based multi-stage programming framework for code generation in c++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–51, 2021.

[5] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 233–244, New York, NY, USA, 2009. Association for Computing Machinery.

[6] E. W. Dijkstra. "a note on two problems in connexion with graphs", 1959.

[7] A. Einstein. Die grundlage der allgemeinen relativitätstheorie. *Annalen der Physik*, 354(7):769–822, 1916.

[8] Ernesto Estrada and Juan A. Rodríguez-Velázquez. Subgraph centrality and clustering in complex hyper-networks. *Physica A: Statistical Mechanics and its Applications*, 364:581–594, may 2006.

[9] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[10] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 319–333, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), nov 2019.

[12] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, 2015.

[13] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[14] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, September 2009.

[15] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[16] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[17] Jure Leskovec and Rok Sosic. SNAP: A general purpose network analysis and graph mining library. *CoRR*, abs/1606.07550, 2016.

[18] E. F. Moore. "the shortest path through a maze". *Proceedings of an International Symposium on the Theory of Switching*, page 285–292, April 1957.

[19] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[20] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A high-performance sparse tensor algebra compiler in multi-level ir, 2021.

[21] Paul Tune, Hung X. Nguyen, and Matthew Roughan. Hidden markov model identifiability via tensors. 2013.

[22] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.