SPARSE FORMAT CONVERSION AND CODE SYNTHESIS

by

Tobi Goodness Popoola

A dissertation

submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Computing

Boise State University

May 2023

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the dissertation submitted by

Tobi Goodness Popoola

Dissertation Title: Sparse Format Conversion and Code Synthesis

Date of Final Oral Examination: 24 February 2023

The following individuals read and discussed the dissertation submitted by student Tobi Goodness Popoola, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Catherine RM. Olschanowsky, Ph.D.	Chair, Supervisory Committee
Elena Sherman, Ph.D.	Member, Supervisory Committee
Tim Andersen, Ph.D.	Member, Supervisory Committee

The final reading approval of the dissertation was granted by Catherine RM. Olschanowsky, Ph.D., Chair of the Supervisory Committee. The dissertation was approved by the Graduate College.

Dedicated to Dad and Mum

ACKNOWLEDGMENTS

The author wishes to express gratitude to his advisor, Catherine Olschanowsky, for her unwavering patience, encouragement, and support.

This work has been partially supported by the National Science Foundation under Grant Numbers 1422725, 1563818, and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program under Award Number DE-SC-04030, and contract number DE-AC02-05-CH11231. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Department of Energy.

ABSTRACT

Sparse computations are important in scientific computing. Many scientific applications compute on sparse data. Data is said to be sparse if it has a relatively small number of non-zeros. Sparse formats use auxiliary arrays to store non-zeros, as a result, the contents of auxiliary arrays are not known until run-time. The Inspector/Executor (I/E) paradigm uses run-time information for compiler optimizations. An inspector computes information at run-time to drive transformations. The executor—a compile-time transformation of the original code— uses information computed by the inspector. The sparse polyhedral framework (SPF) encompasses a series of tools to support I/E run-time transformations. This work introduces a unified framework that wraps SPF tools while providing a holistic view of computation as an intermediate representation (IR). This work also introduces a method to automatically synthesize inspectors to transform between sparse formats and improvements to SPF to explore the performance of irregular applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ECP – Exascale Computing Project

DOE – Department Of Energy

RAW – Read After Write

WAR – Write After Read

WAW – Write After Write

RAR – Read After Read

AST – Abstract Syntax Tree

IR – Intermediate Representation

COO – Coordinate Format

MCOO – Morton Ordered Coordinate Format

CSR – Compressed Sparse Row

CSC – Compressed Sparse Column

CSF – Compressed Sparse Fiber

ELL – ELLPack Format

BSR – Block-compressed Sparse Row

CSB – Compressed Sparse Block

HiCOO – Hierarchical Coordinate Format

SPF – Sparse Polyhedral Framework

PDFG – Polyhedral+Dataflow Graph

SPMV – Sparse Matrix Vector Multiplication

API – Application Programming Interface

DAG – Directed Acyclic Graph

CFG – Control Flow Graph

DFG – Dataflow Graph

SSA – Static Single Assignment

DSL – Domain Specific Language

PDFG – Polyhedral+Dataflow Graph

PEP – Polyhedral Expression Propagation

SCoP – Static Control Part

CHAPTER 1

INTRODUCTION

Computationally intensive applications such as molecular thermodynamics, climate modeling, and machine learning require significant computational resources. Energy consumed by large data centers running these applications can easily power up small cities. Improving the performance and efficiency of these applications is important. Improved performance means less time spent using computing resources, consequently resulting in better energy consumption. The biggest opportunity for improving performance in scientific applications is reducing memory movement. Arithmetic computations are typically fast and bandwidth to memory is the bottleneck. The processor must wait to get data from memory, this degrades performance. The further away memory is from the CPU, the more computational cycles are required to move data during a computation. This work aims to reduce memory movement in applications, specifically focused on applications operating on sparse data. We achieve this through temporary storage reduction, memory expansion, and remapping, and graph simplification algorithms/heuristics to automatically transform computations to better use memory.

Optimizing computations that run on sparse data is complicated due to indirect memory accesses. These types of applications are referred to as irregular applications. A data set is said to be sparse if it has a relatively small percentage of data values that are not zero. To save memory, sparse formats store only non-zero values and use auxiliary arrays to identify the original coordinates of the non-zeros. The auxiliary arrays and the fact that the sparsity pattern is not known until run-time limit static transformation. The Inspector/Executor paradigm is a common technique for optimizing irregular applications. This entails writing compiled code that will make decisions at run-time, Inspector, and transforming the original computation to use the inspector, Executor. However, manually writing this code by hand is tedious and error-prone.

Automatic optimization of regular applications uses tools that are not applicable to irregular applications. These tools do not support the Inspector/Executor paradigm and other relevant run-time reorder transformations. Various tools such as Omega/Codegen, and IEGenLib do provide a framework for transforming irregular applications, however, there is the need to provide a uniform entry point for this toolchain. This work introduces a unified framework that wraps relevant sparse tool chains while providing a holistic view of computation. This framework supports composable transformations, data flow representation and analysis, and a front end to automatically translate legacy applications to a representation suitable for transformation.

There are several formats used to store sparse data. The optimal choice of sparse format for computation depends on the data and the computation. Deciding the best sparse format for computation is not the focus of this work, rather it focuses on enabling transformations post-decision. A recent work [29] shows how the choice of matrix format affects Graph Neural Network (GNN) performance. They developed a predictive model that can dynamically choose the optimum sparse format to be used by a GNN layer depending on the input matrices. This shows that there is no one-size-fits-all for the choice of sparse formats, hence the need to transform from one format to another (Synthesis). Synthesizing format conversion code that is performant is preferable to handwriting and optimizing all possible combinations.

The best choice of sparse tensor format changes with computational patterns and the sparsity pattern of data. Once a choice of format has been made, optimized routines that transform the sparse code from one format to another are required. Sparse format conversion can be from any sparse format to any other sparse format, creating a vast space of transformations. Furthermore, that choice may change over the lifetime of application execution. As a simple example, consider a sparse tensor that is used in multiple phases of computation and will sometimes be read in the first mode and later in the last. Changing formats between phases may be advantageous depending on the number of times the operations are executed. We demonstrate the expressiveness of the specification with a collection of common sparse tensor formats and we evaluate the correctness of the synthesis algorithm. This work presents an approach that describes sparse tensor formats and synthesizes translation code among them.

1.1 Contributions

Several research questions remain to be answered before achieving effective optimization of scientific applications that use sparse data. The first question reasons about what constitutes a complete internal representation for sparse computations and how that internal representation can be transformed using known methods. Given the lack of required information at compile-time, a second research question is on how data transformations can be automated without breaking the abstractions in the internal representation. This work introduces two research contributions to answer these questions.

1. Expose composable sparse polyhedral transformations through an object-oriented API. The result is a well-specified intermediate representation for describing and transforming sparse computations.

2. Synthesize inspectors for sparse format conversion and co-iteration optimizations. This introduces an automation technique for transforming the data layout of a sparse computation.

1.2 Dissertation Structure

This dissertation is structured as follows. Chapter 2 introduces the necessary background required to understand this material, the compilation pipeline, polyhedral model, sparse polyhedral model, sparse formats, and inspector/executor paradigm. Chapter 3 introduces the SPF intermediate representation and its supported transformations. Chapter 4 introduces code synthesis, a major contribution of this work where we describe sparse data layouts and use constraint relationships in transforming from one layout to another. The related work to this material is provided in Chapter 5 and Chapter 6 contains the conclusion and future work.

CHAPTER 2

BACKGROUND

In this section, we discuss concepts that form the basis of our work. The compiler pipeline, polyhedral model, sparse polyhedral model, and sparse formats are key concepts used in our research.

2.1 Compiler Pipeline

A typical compiler pipeline consists of four stages: Preprocessor, Compiler, Assembler and Linker as shown in Figure 2.1

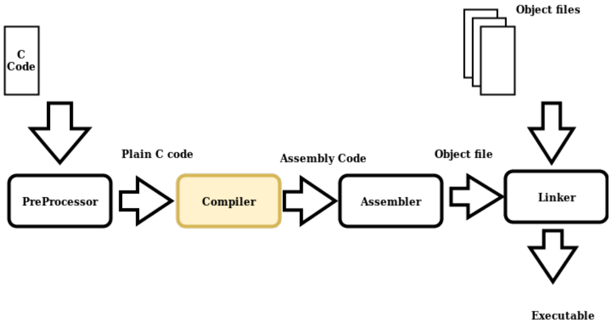


Figure 2.1: Compiler Pipeline

The **preprocessor** reads a high-level language containing possible directives, processes the directives, and produces source code. The **compiler** produces low-level assembly code from a high-level programming code. The **assembler** generates an object file from a low-level assembly code. Finally, the **linker** combines the object file with other object files and the run-time libraries to produce a binary executable. We focus on transformations that are applied during the compilation stage

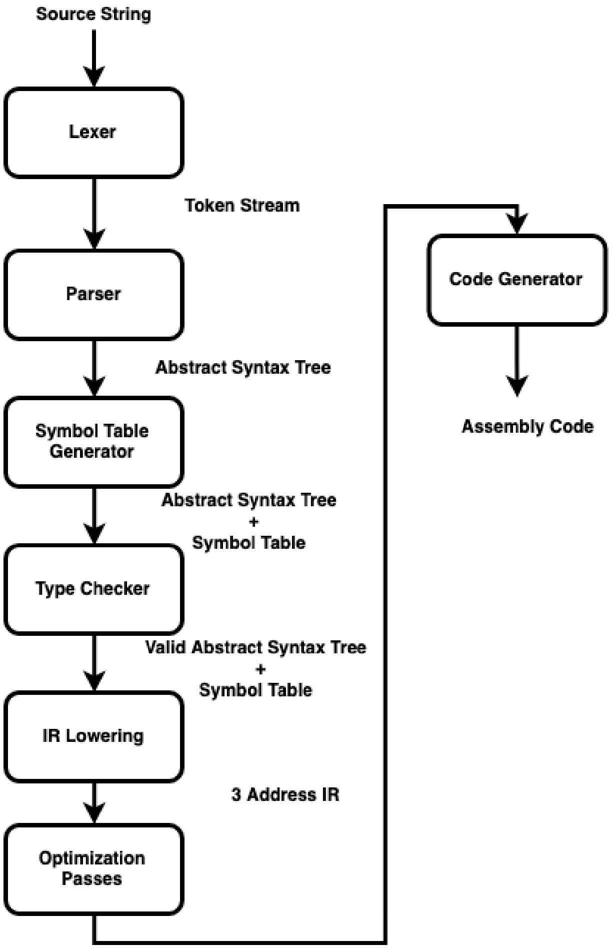


Figure 2.2: Compiler Stages

The compilation is divided into various stages, the first stage tokenizes the source text into lexicons, this stage is important as it extracts important parts of the source code sent to the parser. The parser is the second stage of the compiler pipeline, in this stage, the Abstract Syntax Tree (AST) of the program is built from the stream of tokens using a context-free grammar. Grammars define the syntax of a high-level programming language. The abstract syntax tree shows the syntatic structure of the programming language text.

The next stage involves building the symbol table, which is a data structure that keeps information on identifiers, and functions (formal parameter signature, return type, and other information on the function). Type checking involves the compiler checking if the program written by the user adheres to typing rules provided by the high-level language, the symbol table plays an important role in type checking as the compiler could access important information during this process.

Optimizing compilers usually have an intermediate representation that is generated from the AST of the program. Intermediate Representations or IR could expose various optimization opportunities. Clang /LLVM use the LLVM-IR as an intermediate representation and GCC uses Gimple. LLVM-IR [33] uses the RISC-like instruction format (also known as three address code) of the language to enable generic low-level optimizations and the typing information allows for high-level optimizations. The LLVM-IR [33] and Gimple [1] use single static assignment (SSA), which means a variable is only written to once and could be read multiple times in the program. The AST and other lower-level IRs earlier discussed provide different optimization opportunities.

Assembly Code generation for target machines indicates the end of the compilation pipeline.

2.2 Polyhedral Model

The polyhedral model is an effective optimization tool for applications with affine loop bounds. This model represents computations as sets and relations. Iteration spaces are represented with sets and data dependences are represented using relations. This combination of iteration spaces and data dependences provides a partial ordering for the target computation. Within the partial ordering, the order of execution can be altered by enforcing relations to the iteration space. These relations are referred to as transformations.

```c
for(int i=0; i<M; i++){
  for(int j=0; j<N ;j++){
    printf("i: %d, j:%d\n",i,j)
  }
}
```

Figure 2.3: A simple nested loop

2.2.1 Affine Sets and Iteration Domain

The iteration domain represents the instances of a statement in a computation. A presburger set denotes the iteration domain. A presburger set is a set with a notation that includes a structured integer tuple template and a set builder notation constraint (presburger formula). Variables constrained in the presburger formula could consist of variables in the template and symbolic constants.

$$\{[i,j] \in \mathbb{Z}^2 : 0 \leq i < n \wedge 0 \leq j < n : n \in \mathbb{Z}\} \tag{2.1}$$

In Equation 2.3, $[i,j]$ is the structured integer tuple template formalised as shown in Equation 2.2 and n is a symbolic constant introduced in the constraints.

$$[i_0, i_1, ..., i_{d-1}], d \geq 0 : d, i \in \mathbb{Z} \tag{2.2}$$

The presburger formula in Equation 2.3 example is given as $0 \leq i < n \wedge 0 \leq j < n$. The presburger set can be fully expanded as a roster notation for $n = 3$ as shown below

$$\{[0,0],[0,1],[0,2],.....[2,2]\}$$

The polyhedral model uses this reasoning to represent iteration instances as presburger sets. Take for example the loop shown in Figure 2.3 can be represented as the presburger set below

$$\{[i,j] : 0 \leq i < M \wedge 0 \leq j < N\}$$

2.2.2 Affine Access Relations, Maps, and Schedules

A map is a binary relation of a structured integer tuple. Maps are formally denoted as the presburger relation- element pair description consisting of a pair of templates and a presburger formula C in terms of the variables in the template.

$$\{[i_0,..,i_d] \in \mathbb{Z}^d \to [j_0,...,j_n] \in \mathbb{Z}^n : C\} \tag{2.3}$$

Access relations map the iteration domain of a statement to its array accesses. The access relation map is important for data dependence analysis and validating transformations. Read-after-write (RAW) and write-after-read (WAR) dependencies have to be maintained after transformations. On the other hand, write-after-write (WAW) and read-after-read (RAR) dependencies do not need to be preserved.

The Figure 2.4 shows S1 and S2 reading from arrays a and t respectively expressed

```
for(int i=0; i<N; i++){
S1: t[i] = f(a[i])
}
for(int i=0; i<N ;i++){
S2: b[i] = g(t[N-i-1])
}
```

Figure 2.4: A code listing showing reads and writes on array accesses.

as an access relation shown below

$$\{S1[i] \rightarrow a[i]\}$$

$$\{S2[i] \rightarrow t[N-i-1]\}$$

Statements S1 and S2 writes to t and b respectively as shown below

$$\{S1[i] \rightarrow t[i]\}$$

$$\{S2[i] \rightarrow b[i]\}$$

A Schedule is a strict partial order of elements on the instance set that specifies the order in which statements should be executed. Statement S1 is executed before statement S2. This information can be represented with the schedule.

$$\{S1[i] \rightarrow [0, i]\}$$

$$\{S2[i] \rightarrow [1, i]\}$$

Maps can be used to transform the iteration domain of a statement to explore optimization opportunities. One can apply the following relation to the iteration space of the code listing in Figure 2.3 to do loop interchange.

$$IC = \{[i, j] \to [j, i]\}$$

$$I' = IC(I)$$

Performing code generation on I' yields the code listing in Figure 2.1.

```
for(int j=0; j<N; j++){
 for(int i=0; i<M ;i++){
  printf("i: %d, j:%d\n",i,j);
 }
}
```

Listing 2.1: Transformed loop after applying Interchange (see Figure 2.3).

2.2.3 Affine Operations

The polyhedral model supports some operations that are used in this work. The operations include cardinality, affine hull, range of a map, union, intersection, apply map to set, equality, subset and super set to mention a few.

2.3 Sparse Polyhedral Model

The sparse polyhedral framework (SPF) extends the polyhedral model by supporting non-affine iteration spaces and transformations using *uninterpreted functions*. SPF provides much of the same functionality as traditional polyhedral tools: code generation with CodeGen+ [15] built on Omega [57] and precise set and relation operations in the presence of uninterpreted functions with IEGenLib [51].

2.3.1 Uninterpreted Functions

In sparse computations, array accesses become part of the loop bounds in computation, this cannot be modeled in the polyhedral framework. This is termed non affine and is represented as uninterpreted functions in the sparse polyhedral model. Uninterpreted functions (UF) are a special case of symbolic constants. Uninterpreted functions represent data structures such as index arrays in sparse data formats. A unique advantage of introducing uninterpreted functions is that mathematical properties of a function can be added to the model thereby extending the applicability of the polyhedral framework. For example every function satisfies the constraint that if the same input is given, the same output will be produced. This is called functional consistency, defined below

$$z = x \implies f(z) = f(x)$$

SPF provides mechanisms to further describe uninterpreted functions- domain, range, and index array properties. This information is used for data dependence optimizations [36] and in this work, for code synthesis.

2.3.2 Non-Affine Iteration Space

When defining the space for a sparse matrix-vector multiplication in the code listing in Figure 3.2, non-affine array accesses used in loop bounds are modeled as uninterpreted functions as shown below

$$I = \{[i, k] : 0 \leq i < N \;\wedge\; rowptr(i) \leq k < rowptr(i+1)\}$$

2.3.3 Non-affine Data Accesses and Maps

In SPF, Uninterpreted functions are allowed in data access relations. The Sparse Matrix Vector Multiplication (SPMV) as shown in Figure 3.2 show an indirect read access to the vector x modelled as a relation shown below

$$\{[i, k] \rightarrow [t] | t = col(k)\}$$

2.3.4 Non-affine Transformations

Transformations done in the sparse polyhedral framework are built on the inspector executor paradigm. Uninterpreted functions can be introduced into a transformation and applied to a computational kernel (executor) with the assumption that the uninterpreted functions will already be generated at compile time and populated at run-time (inspector).

2.3.5 Composition Theorems

Strout et al. 's [52] work introduced composition theorems used in this work. Their work introduced precise application and composition operations in non-affine sets and maps. Proofs for the listed theorems can be found in the referenced work [52]. In this work, we make use of Case 1 theorem 1

Theorem 1 (Case 1: Both Relations are functions) *Let $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{v}$, and $\boldsymbol{z}$ be integer tuples where $|\boldsymbol{y}| = |\boldsymbol{v}|$, $F1()$ and $F2()$ be either affine or uninterpreted functions, and C1 and C2 be sets of constraints involving equalities, inequalities, linear arithmetic, and uninterpreted function calls in*

$$\{\boldsymbol{v} \rightarrow \boldsymbol{z} | \boldsymbol{z} = F1(\boldsymbol{v}) \wedge C1\} \circ \{\boldsymbol{x} \rightarrow \boldsymbol{y} \mid \boldsymbol{y} = F2(\boldsymbol{x}) \wedge C2\}$$

The result of the composition is $\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid \exists\ \boldsymbol{y}, \boldsymbol{v} \mid \boldsymbol{y} = \boldsymbol{z} \wedge \boldsymbol{z} = F1(\boldsymbol{v}) \wedge C1 \wedge \boldsymbol{y} = F2(\boldsymbol{x}) \wedge C2\}$ *which is equivalent to*

$$\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid z = F1(F2(\boldsymbol{x})) \wedge C1[v/F2(x)] \wedge C2[y/F2(x)]\}$$

where $C1[v/F2(x)]$ *indicates that* $\boldsymbol{v}$ *should be replaced with* $F2(x)$ *in the set of constraints* $C1$*.*

2.4 Inspector/Executor Paradigm

An inspector computes information at run-time to drive transformations. The executor—- a compile-time transformer of the original code—- uses information computed by the inspector. Inspectors can be reasoned as the population of uninterpreted functions and executors can be reasoned to use these uninterpreted functions to guide transformations.

2.5 Sparse Formats

Sparse formats describe how sparse coordinates and corresponding data are stored and are often based on sparse matrix formats. Data is said to be sparse if it has a relatively large amount of non-zeros. Figure 2.6 shows a few of the most common sparse matrix formats including the coordinate format. The coordinate (COO) format stores each non-zero and stores the coordinate indices in separate arrays organized by dimension. Compressed Sparse Row (CSR) compresses the rows and each non-zero is ordered and has a corresponding uncompressed column coordinate. Blocked Compressed Sparse Row splits the dense matrix into blocks and compresses the blocked rows. Diagonal (DIA) compresses each diagonal of a matrix.

Sparse tensor formats specifically designed for higher dimensional data include HICOO [34] and Alto [28]. These formats are more complex and involve sorting and other data structures. Morton Coordinate (MCOO) format preserves locality of the data points in its multidimensional coordinates. The binary of the coordinates of each non zeros are interleaved and the resulting value sorts the placement of data values in the format.

The key concept for each sparse tensor format is that the auxiliary (or index) arrays provide the dense coordinates of the corresponding data. Taken together they provide a mapping from an iteration space to a data space.

```
for(int i=0; i<N; i++){
  for(int k=rowptr[i]; k < rowptr[i+1] ;k++){
     y[i] += a[k] * x[col[k]];
  }
}
```

Figure 2.5: Sparse Matrix Vector Multiply

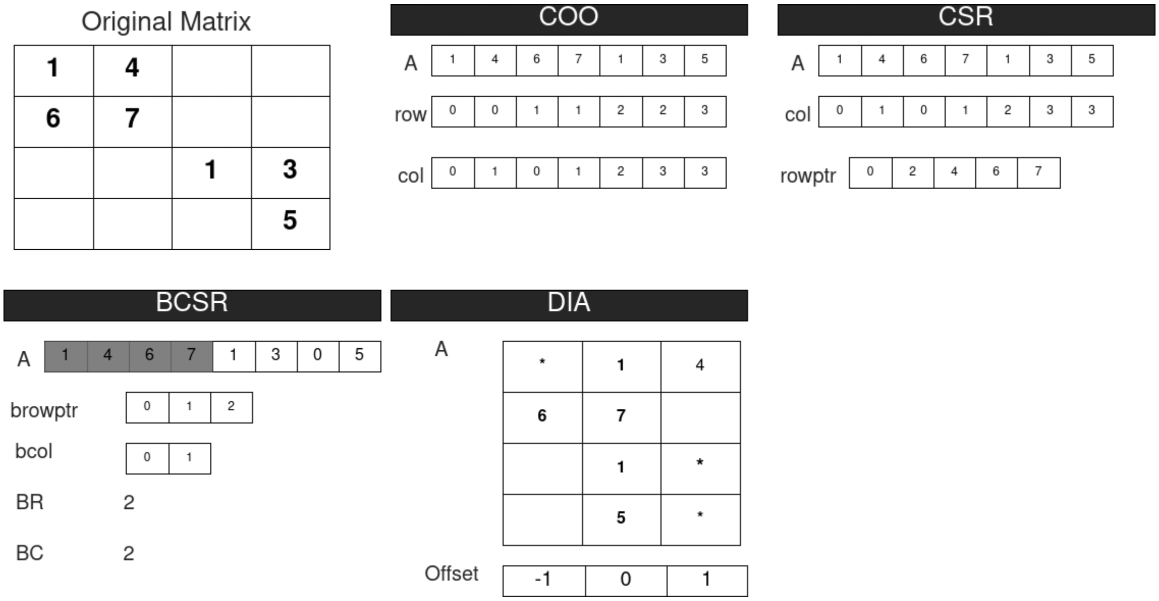


Figure 2.6: Sparse Matrix Formats.

CHAPTER 3

COMPUTATION INTERMEDIATE REPRESENTATION

The sparse polyhedral framework extends the polyhedral model by supporting non-affine iteration spaces and transformations using *uninterpreted functions*. Uninterpreted functions are symbolic constants that represent data structures such as the index arrays in sparse data formats. The SPF provides much of the same functionality as traditional polyhedral tools: code generation with CodeGen+ [15] built on Omega [57] and precise set and relation operations in the presence of uninterpreted functions with IEGenLib [51]. However, these tools are not tightly integrated and each has their own low-level API to interact with individual components. These components include ways to represent sets and relations internally, statements in a computation, codegeneration, and analyses. In some cases some tools support code generation [16, 17, 45, 55] while some others do not [2].

This chapter discusses the SPF intermediate representation (SPF-IR) and the Computation API. The Computation API provides a precise specification of how to combine the individual components of the SPF to create an intermediate representation. This IR can directly produce Polyhedral Dataflow Graphs (PDFGs) [21] and translates graph operations defined for PDFGs into relations used by IEGenLib to perform transformations. In this work, we extend the PDFG representation to handle non-affine loops with imperfect nesting and loop carried dependences.

Figure 3.1 shows an overview of our optimization framework and where the computation API fits in the process. Spf-i/e highlighted grey in the figure is a tool that automates translation from source code to our IR and is publicly available [5]. In this work we focus on manually translating original source code to the SPF IR. The SPF IR by itself supports composable transformations by virtue of the sparse polyhedral framework. Graph operations on a PDFG generates composable transformations and are applied to statements in the IR.

This chapter answers the first research question that reasons about what constitutes a complete internal representation for sparse computations and how that internal representation be transformed using known methods. To answer this question, we introduce a formal intermediate representation using low level SPF API such as CodeGen+, Omega and IEGenlib to provide a unique entry point to describe sparse computations in a manner suitable for transformations. We also provide standard well known operations such as fusion, dead variable elimination, inlining, reschedule and unaffined transformation operations on statements in the representation. CHiLL [27], an SPF tool, provides a script based interface for the SPF and supports non-affine transformations by using Omega. Omega has limitations on the precision of operations involving uninterpreted functions, as well as restrictions on how they must be expressed. This work overcomes limitations of previous approaches by making use of precise set and relation manipulation tool using IEGenlib.

Visualizing a computation as a graph is a well known method in guiding optimizing experts. We further answer this research questions by enabling the automatic generation of a Polyhedral Data Flow Graph (PDFG) to provide research experts to better reason about optimizations.

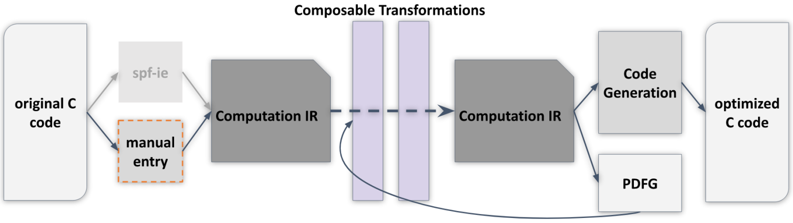


Figure 3.1: Optimization Pipeline Overview

3.1 Computation: A C++ Class

The interface to the SPF is implemented as a C++ class in IEGenLib. The computation class contains all of the information required to express a computation or a series of computations. This includes: data spaces, statements, data dependences, and execution schedules. This section describes the design and interface for each of these elements. Matrix vector multiplication is used as a running example throughout the rest of this paper. Figures 3.1 and 3.2 show the dense and sparse versions.

```
for (i = 0; i < N; i++) {
    for (j=0; j<M; j++) {
        y[i] += A[i][j] * x[j];
    }
}
```

Listing 3.1: Dense Matrix Vector Multiply

```
for (i = 0; i < N; i++) {
    for (k=rowptr[i]; k<rowptr[i+1]; k++) {
        j = col[k];
        y[i] += A[k] * x[j];
    }
}
```

Listing 3.2: Sparse Matrix Vector Multiply

3.2 Data Spaces

A data space represents a collection of conceptually unique memory addresses. A data space of 0 dimensions is equivalent to a scalar. Each combination of data space name and input tuple is guaranteed to map to a unique space in memory for the lifetime of the data space. For example, $D(\vec{s})$, guarantees that for each unique tuple $\vec{s}$ there will be a unique memory location, and that during the lifetime of D its memory locations will not overlap with any other live data space's memory locations.

By default all data spaces are generic. They are defined with the syntax $D(\vec{s})$. For example, for a 3 parameter input tuple i, j, k the data space can be represented as $D(i, j, k)$. This data space can be written to only once but read from any number of times. The exception to this rule is for accumulation operations when a single data location within a data space can be written to multiple times $(+ =, - =, * = , max, min...)$.

The data spaces represented in the matrix vector multiply example include: y, A, and x. In the sparse version the index arrays *rowptr* and *col* are also considered data spaces. However, since they are used within the loop bounds and to access into another data space they must be constant for the duration of this computation. Therefore, they are not required as part of the Computation's definition.

```
// dense
Computation* dsComp = new Computation();
dsComp->addDataSpace("y");
dsComp->addDataSpace("A");
dsComp->addDataSpace("x");

// sparse
Computation* spsComp = new Computation();
```

```
spsComp->addDataSpace("y");
spsComp->addDataSpace("A");
spsComp->addDataSpace("x");
```

3.3 Statements

Statements perform read and write operations on data spaces. We restrict the definition of statements to be basic blocks. There is a single entry and a single exit from each block of code represented.

All statements have an iteration domain associated with them. This iteration domain is a set containing every instance of the statement and has no particular order. It is typically expressed as the set of iterators that the statement runs over, subject to the constraints of their iteration (loop bounds). The following code block shows how to create a statement using the Computation API. A statement is written as a string as seen on lines 2 and 5 below for the dense and sparse cases respectively. The iteration domain is specified as a set using the IEGenLib syntax, with the exception of delimiting all data spaces with $, this can be seen on lines 3 and 6 below.

```
1 Stmt* ds0 = new Stmt(
2   "y(i) += A(i,j) * x(j);",
3   "{[i,j]: 0 <= i < N && 0 <= j < M}", ...
4
5 Stmt* sps0 = new Stmt(
6   "y(i) += A(k) * x(j)",
7   "{[i,k,j]: 0 <= i < N && rowptr(i) <= k < rowptr(i+1) && j = col(k)}", ...
```

3.4 Data Dependence Relationships

Data dependences exist between statements. They are encoded using relations between iteration vectors and data space vectors. Calculating a closure provides the dependence relationships between statements and a partial ordering constraint on the calculation. In our running examples the data reads and writes can be specified as written below.

```
/* 4th and 5th parameters to Stmt constructor */
// dense
...
{ // reads
  {"y", "{[i,j]->[i]}"},
  {"A", "{[i,j]->[i,j]}"},
  {"x", "{[i,j]->[j]}"}
},
{ // writes
  {"y", "{[i,j]->[i]}"}
}

// sparse
...
{ // reads
  {"y", "{[i,k,j]->[i]}"},
  {"A", "{[i,k,j]->[k]}"},
  {"x", "{[i,k,j]->[j]}"}
},
{ // writes
  {"y", "{[i,k,j]->[i]}"}
}
```

3.5 Execution Schedules

Execution schedules are determined using scattering functions that are required to respect the data dependence relations. Scheduling functions take as input the iterators that apply to the current statement, if any, and output the schedule as an integer tuple that may be lexicographically ordered with others to determine correct execution order of a group of statements. Iterators are commonly used as part of the output tuple, representing that the value of iterators affects the ordering of the statement. For example, in the scheduling function $\{[i, j]- > [0, i, 0, j, 0]\}$, the position of i before j signifies that the corresponding statement is within a loop over j, which in turn is within a loop over i. Additionally, in a lexicographical ordering, all instances of the statement with $i = 1$ will precede all instances with $i = 2$, regardless of the value of j.

```
/* 3rd parameter to the Stmt constructor */
// dense
 "{[i,j] ->[0,i,0,j,0]}"

 // sparse
"{[i,k,j]->[0,i,0,k,0,j,0]}"
```

Figure 3.2 shows the complete specification of two computation, first dense matrix vector multiply followed by sparse matrix vector multiply.

3.6 Code Generation

The Computation class interfaces with CodeGen+ [15] for code generation. CodeGen+ uses Omega sets and relations for polyhedra scanning. Omega sets and relations have limitations in the presence of uninterpreted functions. Uninterpreted functions are limited by the prefix rule. This rule states that an uninterpreted function must be a prefix of the tuple declaration. Uninterpreted functions cannot have expressions as parameters. Code generation overcomes this limitation by modifying uninterpreted functions in IEGenLib to be Omega compliant, while storing a mapping of the original uninterpreted function to its modified uninterpreted function. The separation of representations for transformations and code generation allows precise operations during transformations while still leveraging the functionality of CodeGen+ for polyhedra scanning.

Figure 3.3 shows the results of code generation for the sparse matrix vector multiplication computation defined in Figure 3.2. Line 2 of Figure 3.3 defines a macro for the statement s0, lines 9 - 13 remap the Omega compliant uninterpreted function back to its original. Lines 15 - 20 are a direct result of polyhedra scanning from CodeGen+. The Computation implementation provides all of the supporting definitions for fully functional code.

3.7 Visualizing a Computation on a Graph

Visualizing computations as a data flow graph gives performance experts a suitable view to reason about transformations for optimization opportunities. To this end, PDFGs express regular computations using a combination of the polyhedral model and dataflow graphs [21]. The original graphs have certain limitations: loop carried dependences were not expressed, and imperfect loop nests were not supported. This section describes how these limitations were overcome. Additionally, the creation of PDFGs by traversing the SPF IR is integrated with the Computation API. After a

Computation is created such as the one in Figure 3.2 a function can be called that outputs the PDFG as a dot file.

In the original PDFG, shaded rectangular boxes represent data spaces and inverted triangles represent statements. In the extended PDFG, shaded rectangular boxes represent domains, transparent rectangular boxes represent data spaces and rounded rectangular boxes represent statements. Edges represent reads and writes in both the original and extended PDFG. The extended PDFG does not currently express the type and size of data spaces in a computation.

Loop carried dependences and imperfect loop nests are important patterns to consider when deciding which optimizations to apply. Loop carried dependences refer to coding patterns where one iteration of a loop reads or writes data produced by another iteration of the same loop. An imperfect loop nest is one that has statements at multiple levels as shown in Figure 3.4.

3.8 Loop Carried Dependences

The existing representation is not capable of visualizing the presence of a loop carried dependence. Figure 3.6 shows the PDFG for the forward solve example in Figure 3.4. In the example, the code contains a loop carried dependence for the data space u in statements $S1$ and $S2$. However, the original PDFG graph in Figure 3.6 does not visualize the loop carried dependence.

Figure 3.5 shows the extended PDFG. The outgoing edge from the data node u to the statement node $S1$ at index j shows the read access in the j-loop. The incoming edge from the statement node $S2$ to the data node u at index i shows the write access in the i-loop. This shows a loop carried dependence over i.

3.9 Imperfect loop nests

The forward solve example also exhibits an imperfect loop nest pattern: statements $S0$ and $S2$ are in the outer i-loop while statement $S1$ is in the inner j loop. The original PDFG in Figure 3.6 does not exhibit this pattern. The updated design in the extended PDFG show in Figure 3.5 visualizes the imperfect loop pattern.

3.10 Operations on Computations

Computations are composed of IEGenLib Sets and Relations. IEGenLib's functionality is used directly to manipulate the Computations' components. The core operations implemented in IEGenLib include inversion, compose, apply, and related supporting functions. Using these operations we are able to perform function inlining and loop transformations within the Computation class.

3.11 Inlining

Computation inlining handles the complexity of creating the SPF representation of functions that call other functions. Instances of Computations need to be reusable in the same way that functions are reusable.

Each function in the original source code is represented as a Computation. When a function call is first encountered in the source, a Computation must be created for it. Then, the contents of that Computation are inserted (inlined) into the caller's Computation that is being constructed. If the same function is called multiple times, the Computation that has been generated for it will be reused. The inlining process can continue to any nesting depth, if a function being called also calls other functions.

The inlining function is responsible for:

- avoiding naming conflicts between variables in the caller and callee,
- generating assignment statements for function parameters,
- providing callee return values to the caller and
- updating iteration domains, execution schedules and data dependences in the inlined statements.

Before inlining, the names of data spaces and iterators in the callee are prefixed with a unique string to avoid collisions with the caller's data spaces, or with other instances of the same inlined Computation within the same scope. This change is reflected in the stored source code string of the statement, as well as other parts of its representation that involve these names.

To keep things simple, the return values and arguments to an inlined function are restricted to either names of data spaces or literals. To pass a more complicated expression into a function, like `A[0]` or `x+y`, it must first be assigned to a temporary variable which can then be passed in. To preserve the original calling semantics of the program, when arguments are passed to a function, statements are generated to declare each of its parameters equal to the passed in values. No equivalent process occurs with return values, because they could potentially be used in a larger variety of contexts (assigned to variables, used immediately in an expression, or ignored entirely). Therefore, the inlining process returns the values that are returned by the inlined function, as strings, to be used however the caller sees fit.

Iteration domains, execution schedules, and data access relations are updated to reflect the surrounding context the statements have been inserted into. For example, if a function is called within a loop, and the callee itself also contains a loop, the representation of the innermost statements are adjusted to reflect that they are now nested under both loops.

3.12 Loop Transformations as Graph Operations

In this work, we provide fusion and reschedule operations on the graph. The operations on the graph translates to transformations performed by our interface API to manipulate the SPF components.

3.12.1 Fusion

This is a transformation that joins two statements from separate loops into a single loop. There are various categories of loop fusion, including *read reduction fusion* and *producer-consumer fusion*. Read reduction involves fusing loops that read from the same memory location while producer-consumer fusion involves merging loops where one loop writes a variable that is then read by the second loop.

Graph operation for fusing two statements together at a particular level of the execution schedule is denoted as $fuse(S1, S2, level)$ in our IR. $S1$ and $S2$ are the statements to be fused and *level* indicates what depth to fuse at. Specifying the depth to fuse allows for more flexibility in a fusion operation. After fusion, $S2$ will be ordered immediately after $S1$.

3.12.2 Reschedule

The reschedule operation involves moving a statement to a new location in the graph and consequently changing its execution schedule. Rescheduling by itself is not an optimization, however, it exposes optimization opportunities. The reschedule graph operation is denoted as $reschedule(S1, S2)$ in our IR. This will cause statement $S1$ to be rescheduled to appear before $S2$.

3.12.3 Dead Variable Elimination

Eliminating redundant computations is an optimization technique that improves the performance of an application. In a graph, a statement node is dead if it writes to a data node that is never read from. In our work, we provide this transformation as an option to the Computation intermediate representation. It should be noted that this operation removes dead assignments from the Computation intermediate representation. We implement this functionality by performing a breadth-first search from dead data nodes in the graph, we keep removing statement nodes recursively until we reach data nodes that are read from by other statements. A breadth-first approach allows our algorithm to remove dead assignments per each level when traversing the graph backward. This operation results in a significantly smaller dataflow graph. Code that appears dead but have side effects are not eliminated. This includes code that write to data locations marked as function parameters and memory locations that are aliased.

```
// dense mvm
Computation* dsComp = new Computation();

// add data spaces
dsComp->addDataSpace("y");
dsComp->addDataSpace("A");
dsComp->addDataSpace("x");

Stmt* ds0 = new Stmt(
  // source code
  "y(i) += A(i,j) * x(j);",
  // iter domain
  "{[i,j]: 0 <= i < N && 0 <= j < M}",
  // scheduling function
  "{[i,j] ->[0,i,0,j,0]}",
  { // data reads
    {"y", "{[i,j]->[i]}"},
    {"A", "{[i,j]->[i,j]}"},
    {"x", "{[i,j]->[j]}"}
  },
  { // data writes
    {"y", "{[i,j]->[i]}"}
  }
);
dsComp->addStmt(ds0);

// sparse mvm
Computation* spsComp = new Computation();

// add data spaces
spsComp->addDataSpace("y");
spsComp->addDataSpace("A");
spsComp->addDataSpace("x");

Stmt* sps0 = new Stmt(
  "y(i) += A(k) * x(j)",
  "{[i,k,j]: 0 <= i < N && rowptr(i) <= k < rowptr(i+1) && j = col(k)}",
  "{[i,k,j]->[0,i,0,k,0,j,0]}",
  {
    {"y", "{[i,k,j]->[i]}"},
    {"A", "{[i,k,j]->[k]}"},
    {"x", "{[i,k,j]->[j]}"}
  },
  {
    {"y", "{[i,k,j]->[i]}"}
  }
);
spsComp->addStmt(sps0);

```

Figure 3.2: Computation API specification for dense and sparse matrix vector multiply

```
#undef s0
#define s0(__x0, i, __x2, k, __x4, j, __x6)   y(i) += A(k) * x(j)

#undef col(t0)
#undef col_0(__tv0, __tv1, __tv2, __tv3)
#undef rowptr(t0)
#undef rowptr_1(__tv0, __tv1)
#undef rowptr_2(__tv0, __tv1)
#define col(t0) col[t0]
#define col_0(__tv0, __tv1, __tv2, __tv3) col(__tv3)
#define rowptr(t0) rowptr[t0]
#define rowptr_1(__tv0, __tv1) rowptr(__tv1)
#define rowptr_2(__tv0, __tv1) rowptr(__tv1 + 1)

for(t2 = 0; t2 <= N-1; t2++) {
  for(t4 = rowptr_1(t1,t2); t4 <= rowptr_2(t1,t2)-1; t4++) {
    t6=col_0(t1,t2,t3,t4);
    s0(0,t2,0,t4,0,t6,0);
  }
}

#undef s0
#undef col(t0)
#undef col_0(__tv0, __tv1, __tv2, __tv3)
#undef rowptr(t0)
#undef rowptr_1(__tv0, __tv1)
#undef rowptr_2(__tv0, __tv1)
```

Figure 3.3: SPMV codegen

```
for (i=0; i<N; i++){
   S0: tmp(i) = f(i);
   for(j=0; j<i; j++){
      S1: tmp(i) -= A(i, j)*u(j);
   }
   S2: u(i) = tmp(i)/A(i, i);
}
```

Figure 3.4: Forward Solve

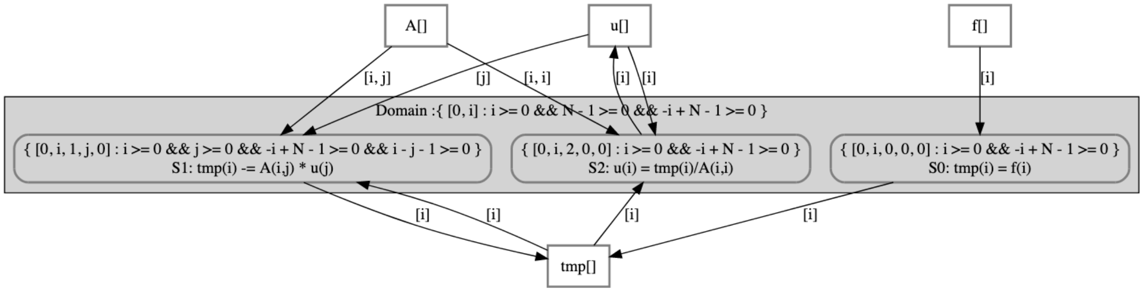

A[]
u[]
f[]
[i, j]
[j]
[i, i]
[i]
[i]
[i]
Domain :{ [0, i] : i >= 0 && N - 1 >= 0 && -i + N - 1 >= 0 }
{ [0, i, 1, j, 0] : i >= 0 && j >= 0 && -i + N - 1 >= 0 && i - j - 1 >= 0 }
S1: tmp(i) -= A(i,j) * u(j)
{ [0, i, 2, 0, 0] : i >= 0 && -i + N - 1 >= 0 }
S2: u(i) = tmp(i)/A(i,i)
{ [0, i, 0, 0, 0] : i >= 0 && -i + N - 1 >= 0 }
S0: tmp(i) = f(i)
[i]
[i]
[i]
[i]
tmp[]

Figure 3.5: Extended PDFG. This graph is automatically generated from the Computation class and includes loop carried dependences and irregular loop nests. The lexicographical ordering of the tuples in the set attached to each statement informs its execution order.

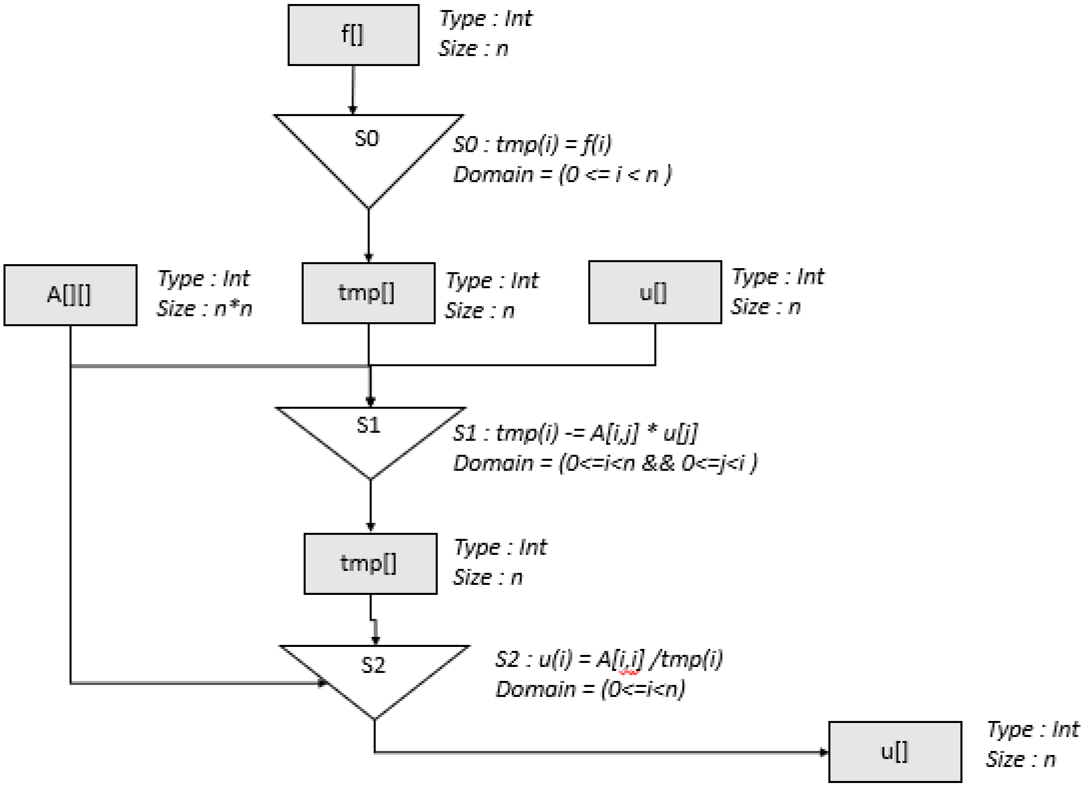


Figure 3.6: The Original PDFG for the Forward Solve example.

CHAPTER 4

CODE SYNTHESIS

Many scientific applications process sparse data. Data is sparse if it has a relatively large percentage of zeros. These applications use sparse tensor formats to reduce memory requirements. Given the increasing number of sparse tensor formats and the need for highly optimized routines that translate among them, automated methods are required to synthesize the translation code. This chapter presents an approach that describes sparse tensor formats and synthesizes translation code among them.

Synthesizing format conversion code that is performant is preferable to handwritting all possible combinations. Further optimizing hand written codes for format conversion is tedious and hard to maintain. The best choice of sparse tensor format changes with computational patterns and the sparsity pattern of data. Once a choice of format has been made, optimized routines that transform the sparse code from one format to another are required. Sparse format conversion can be from any sparse format to any other sparse format, creating a vast space of transformations. Furthermore, that choice may change over the lifetime of application execution. As a simple example, consider a sparse tensor that is used in multiple phases of computation and will sometimes be read in the first mode and later in the last. Changing formats between phases may be advantageous depending on the number of times the operations are executed.

Current automated techniques are limited by the types of sparse formats that can be expressed and transformed between. Our solution supports formats that require ordering such as ALTO [28] and HICOO [34]. These formats (ALTO and HICOO), utilize Morton ordering on the data indices to improve locality when performing mode-agnostic computations. Other approaches to synthesizing sparse format conversion do not support these formats [3, 54].

An expressive and precise mechanism to describe existing and develop new sparse tensor formats is a key component in the code synthesis algorithm. We propose to describe sparse tensor formats as functions from the sparse iteration space to the dense coordinates. The functions are expressed as relations and each uninterpreted function is further described by providing the domain, range, and universal constraints for each. Sparse format descriptors are expressed using the sparse polyhedral framework.

The sparse polyhedral framework (SPF) provides the syntax and operations needed to specify sparse formats and synthesize code from those specifications. SPF supports many loop transformations including fusion, skewing, unrolling, tiling, and others. By directly synthesizing the sparse format code to SPF and expressing the original computation in SPF, both can be optimized in tandem.

The SPF Internal Representation (SPF-IR) provides an object-oriented interface to access SPF operations and requirements for a fully specified computation [41]. It can express a wide class of computations including those with imperfect loop nests and loop-carried dependences. This work proposes a sparse format conversion synthesis technique that emits code expressed in the SPF using the SPF-IR.

This chapter answers the second question in this dissertation a second research on how data transformations can be automated without breaking the abstractions in the internal representation. The contributions of this chapter include:

- A specification of sparse tensor formats using the sparse polyhedral framework, and
- A method to synthesize sparse format conversion routines.

We demonstrate the expressiveness of the specification with a collection of common sparse tensor formats and we evaluate the correctness of the synthesis algorithm. A performance comparison between our work and TACO's implementation as shown in Section 4.3 shows that our approach is competitive or outperforms TACO in cases where a comparison was possible.

4.1 Sparse Tensor Format Conversion

This work introduces an approach to inspector program synthesis using deductive reasoning for sparse tensor format conversion. Sparse formats are described using format descriptors. The descriptors are designed to support a variety of sparse tensor formats, specifically those that depend on user-defined sorting. User-defined sorting allows users to specify re-ordering constraints in a sparse tensor description. Format descriptors are combined to create a mapping from one sparse space to another. The map serves as a source of constrained relationships between the source and destination data structures and is the basis of inspector synthesis.

The inspector synthesis algorithm generates an SPF intermediate representation that ensures uninterpreted functions in the destination format are created and satisfy all constraints. The resulting intermediate representation is a sparse loop chain that populates destination uninterpreted functions. The last operation in the sparse loop chain is the copy operation. The initial, complete sparse loop chain, while correct,

```
for(i = 0; i < NR; i++){
   for(j= 0; j< NC; j++){
      print("%d,%d,%d",i,j,A(i,j))
   }
}
```

Listing 4.1: Kernel to print contents of a dense matrix

will often perform poorly. It can be transformed using standard SPF operations to improve performance.

This section covers each of the required components in detail: sparse format descriptions, the synthesis algorithm, and common transformations.

4.1.1 Sparse Format Descriptor

The sparse format descriptor contains sufficient information to create and use a specific sparse format. Each part of the format descriptor is expressed using SPF notation. The components include a map from the sparse to dense iteration space (a relation), a map from the sparse iteration space to the data (a relation), the domain and range of each uninterpreted function, and a list of universal quantifiers that further describe the uninterpreted functions used in the first map.

Sparse to dense map. A relation expresses a function from the sparse iteration space to the dense iteration space. *The input tuple of the relation is the sparse iteration space.* Intuitively, the sparse-to-dense map can be derived from a computation that iterates through the non-zeros in a sparse format. The computation below iterates through a dense matrix of dimensions $NR \times NC$.

The iteration space of a dense matrix computation as shown in Listing 4.1 is given by the set:

```
for(n = 0; n < NNZ; i++){
  i = row(n)
  j = col(n)
  print("%d,%d,%d",i,j,A(n))
}
```

Listing 4.2: Kernel to print coordinate and values of all non zeros in a COO format

$$\{[i,j] : 0 \leq i < NR \wedge 0 \leq j < NC\}$$

The computation 4.2 iterates through a COO matrix and its iteration space is given by the set:

$$\{[n,i,j] : 0 \leq n < NNZ \wedge i = \mathit{row}(n) \wedge j = \mathit{col}(n)\}$$

Similarly, the computation 4.3 iterates through a CSR matrix and its iteration space is given by the set:

$$\{[i,k,j] : 0 \leq i < NR \wedge \mathit{rowptr}(i) \leq k < \mathit{rowptr}(i+1) \wedge j = \mathit{col}(k)\}$$

A sparse to dense map of COO and CSR is shown in Table 4.2 based on how the sparse iteration space maps to the dense coordinate. The sparse-to-dense map must be a function. This is required by inspector synthesis and executor transformations.

Data access relation. The data access relation, also expressed as an SPF relation, maps from the sparse iteration space to the data space. In our example using COO, the relation is $\{[n,i,j] \rightarrow [n]\}$. The iteration space of CSR is $\{[i,k,j]\}$, and its data access relation is $\{[i,k,j] \rightarrow [k]\}$. The data access relation decouples the

```
for(i = 0; i < NR; i++){
  for (k = rowptr(i) ; k < rowptr(i+1); k++){
      j = col(k)
      print("%d,%d,%d",i,j,A(k))
  }
}
```

Listing 4.3: Kernel to print contents of a CSR matrix

iteration space and the data space allowing them to be transformed separately.

Domain and range. The domain and range of each uninterpreted function are required. In the COO example, the domain of each is the same. Notice that the domain and range definitions, in this case, introduce additional symbolic constants.

Universal quantifiers. Universal quantifiers further refine the specification of the sparse format. COO in Table 4.2 has no universal quantifiers while MCOO introduces a universal quantifier to ensure that this COO format is sorted using a Morton order. This is achieved with a user-defined comparison function. It is important to note that functions that appear *only* within universal quantifiers are user-defined and full definitions must be provided.

4.1.2 Synthesis Algorithm

Code synthesis refers to automatically writing the code that transforms data from one sparse format, the source, to another, the destination. Throughout this section, we refer to examples using COO as the source. This process works using any format as the source. However, most sparse tensors are stored in COO and it is the easiest format to explain.

The input to the synthesis algorithm is two sparse format descriptors and the output is an SPF representation of the inspector. The process begins by composing the inverse of the destination sparse to dense map with the source sparse to dense map (see compose definition in [51]).

$$R_{A_{src} \to A_{dest}} =$$
$$(R_{A_{dest} \to A_{dense}})^{-1} \circ R_{A_{src} \to A_{dense}}$$

Using the relation and the universal constraints, we solve for each unknown uninterpreted function and generate an SPF representation of code that generates those uninterpreted functions. The relation that results from the composition is used to generate the data copy code. Below is a summary of the synthesis process followed by a detailed description of each step.

1. Invert destination format and insert permutation function.
2. Compose sparse to dense maps.
3. For each unknown UF, create the SPF representation to populate.
4. For each quantifier q in Universal Quantifiers, UQ create the SPF representation to enforce.
5. Generate the SPF representation for the copy operation.

Invert destination format relation and insert Permutation. The format description specifies a map from the sparse iteration space to the dense iteration space. Inverting the relation switches the input and output tuples. Next, we introduce a temporary uninterpreted function, referred to as the permutation, to ensure inverse maps are functions: $P([input\_tuple]) = [output\_tuple]$. The input and output tuples are tuples of the inverse destination format. The following demonstrates this step when transforming to MCOO.

$$\begin{aligned} {R_{A_{MCOO} \to A_{dense}}}^{-1} = \{&[i, j] \to [n2, ii, jj] | col_m(n2) = jj \wedge \\ &row_m(n2) = ii \wedge P(i, j) = [n2, ii, jj] \wedge \\ &i = ii \wedge j = jj\} \end{aligned}$$

Compose. The relations are composed to realize a single mapping from the source to the destination iteration spaces. Composition in the presence of uninterpreted functions is supported by IEGenLib [51]. The code synthesis process centers around this mapping.

$$\begin{aligned} R_{A_{COO} \to A_{MCOO}} &= {R_{A_{MCOO} \to A_{dense}}}^{-1} \circ R_{A_{COO} \to A_{dense}} \\ R_{A_{COO} \to A_{MCOO}} &= \{[n1, ii, jj] \to [n2, ii, jj] | \\ &\quad jj = col_1(n1) \wedge col_1(n1) = col_m(n2) \wedge \\ &\quad ii = row_1(n1) \wedge row_1(n1) = row_m(n2) \wedge \\ &\quad P(row_1(n1), col_1(n1)) = [n2, ii, jj]\} \end{aligned}$$

Unknown Uninterpreted Functions. The relation that results from composition in the previous step (in our example $R_{A_{COO} \to A_{MCOO}}$) contains a list of constraints. The uninterpreted functions (UF) from the destination format are assumed to be unknown (unknown UF). Known UFs are UFs from the source format, or that have been resolved at some point in synthesis. We solve for each of the unknown uninterpreted functions and synthesize code to populate them (Unknown UFs: row_m, col_m, NNZ, P).

An unknown UF is solved by using its relationship with known information/functions (from the source format) in our composed map. Note that NR and NC do not appear in this list. It is not possible to reliably derive the shape of a matrix from sparse formats. This is because outermost rows or columns may be zero values and the matrix would look smaller than it is. Therefore, we require that variables be available that describe the shape of the tensor.

The constraints associated with each unknown UF are identified. There are potentially more constraints in this list than first anticipated because we must use substitution to find all of the constraints. The list of constraints associated with each unknown uninterpreted function in this example is shown in Table 4.3.

Synthesizing the code requires that we determine a statement to execute along with an iteration space and an execution schedule. There are two decisions to make at this point. First, which order to generate the unknown UFs, and second, what statements and iteration spaces to synthesize.

Consider the general form of the relation from source sparse format to destination sparse format.

$$R_{A_{src} \to A_{dest}} = \{\vec{x} \to \vec{y} \mid C\}$$
$$\vec{x} \in \mathbb{Z}^l,\ \vec{y} \in \mathbb{Z}^r$$

Where $\vec{x}$ is an integer tuple of length l, $\vec{y}$ is an integer tuple of length r, and C is a constraints list. In the cases below UF represents the unknown uninterpreted function and, f and f' represent functions that comprise linear combinations of known uninterpreted functions and symbolic constants. $\vec{u}$ and $\vec{v}$ are integer tuples that are subsets of $\vec{x}$ and $\vec{y}$ respectively.

Constraints from the resulting relation $R_{A_{src} \to A_{dest}}$ are grouped into 5 cases. Cases 1-3 below deal with constraints that use only tuple variables from the input tuple. Cases 4 and 5 deal with constraints that involve both the input and output tuple variables, but the order is swapped. There are additional combinations of operators and relation characteristics that are not considered. It may be that they will need to be added if they are found to exist in sparse tensor formats. However, at this point, we have added cases only for the combinations that exist in current formats.

Case 1 Constraint: $UF(\vec{u}) = f(\vec{u})$

Statement: $UF(\vec{u}) = f(\vec{u})$

Domain: $\{\vec{u} : C\}$, where C is a constraint list from $R_{A_{src} \to A_{dest}}$ after projection.

Case 1 consists of an equality constraint and both the left and right-hand sides take the same tuple ($\vec{u}$) which is a subset of the tuple variables for the input tuple of the original relation. The corresponding statement is an assignment statement. The iteration space for that statement is created by projecting out all tuple variables from the original relation that are not members of $\vec{u}$. None of the constraints in the running example are categorized as case 1.

Case 2 Constraint: $UF(f'(\vec{u})) \leq f(\vec{u})$

Statement: $UF(\vec{u}) = min(UF(\vec{u}), f(\vec{u}))$

Domain: $\{\vec{u} : C\}$, where C is a constraint list from $R_{A_{src} \to A_{dest}}$ after projection.

Case 3 Constraint: $UF(\vec{u}) \geq f(\vec{u})$

Statement: $UF(\vec{u}) = max(UF(\vec{u}), f(\vec{u}))$

Domain: $\{\vec{u} : C\}$, where C is a constraint list from
$R_{A_{src} \rightarrow A_{dest}}$ after projection.

Cases 2 and 3 are inequality constraints where both the left and right-hand side uses $\vec{u}$ which is a subset of the input tuple variables of the original relation. The unknown UF in case 2 has an upper bound of $f(\vec{u})$, which translates to an assignment to the minimum of $f(\vec{u})$. The unknown UF in case 3 has a lower bound of $f(\vec{u})$, which translate to an assignment to the maximum of $f(\vec{u})$. The iteration spaces of both cases 2 and 3 are created by projecting out all tuple variables from the original relation that are not members of $\vec{u}$. Given that $f(\vec{u})$ is a linear combination of other known UFs, for every tuple instance $\vec{u}$, it is necessary to get the min or max to satisfy the inequalities in cases 2 and 3. None of the constraints in this example are categorized as case 2 or 3.

Case 4 Constraint: $UF(\vec{u}) = f(\vec{v})$
Statement: $UF.insert(F(\vec{u}))$
Domain: $\{\vec{u} : C\}$, where C is a constraint list from
$R_{A_{src} \rightarrow A_{dest}}$ after projection.

Case 4 is an equality constraint where the vector $\vec{u}$, used on the left-hand side, is a subset of the input tuple and the vector $\vec{v}$, used on the right-hand side, is a subset of the output tuple. The statement synthesized is an insert call that takes the function $f(\vec{v})$ as a parameter. The vector $\vec{v}$ in UF differentiates Case 4 from Case 1. In our example, the constraints on row_m, col_m, and P are case 4.

Not all of the constraints that qualify as case 4 in the running example can be satisfied immediately. The relations for the constraints on row_m and col_m are not

```
P = new OrderedList(2,1,MORTON(),"<");
for(int c0=0;c0<NNZ;c0++){
  P.insert(row1(c0),col1(c0));
}
```

Listing 4.4: Permutation Reordering Function P

functions. Taken with the universal constraints the relations for P is a function and should be processed first.

$$\{[\vec{u}] \to \vec{v}]|C_{list}\}$$

The resulting relation for P follows.

$$\{[ii, jj] \to [n2, ii, jj]|\}$$

There are no qualifying constraints. However, when we also consider the universal quantifiers there is enough information to create an exact mapping. The code that would be generated from the SPF-IR representing the synthesized code creates a class that will enforce the universal quantifier.

The Listing 4.4 shows the parameters of the list constructor are the input arity, the output arity, the function to use as a comparator, and the desired operation (less than or greater than). It is important to note that an exact mapping is not required. If the transformation was to an unsorted format an arbitrary order will be used (the order of insertion). The most specific mapping that is found is the one chosen for synthesis.

Case 5 Constraint: $UF(\vec{v}) = f(\vec{u})$

Statement: $UF.insert(F(\vec{u}))$

Domain: $\{\vec{u} : C\}$, where C is a constraint list from
$R_{A_{src} \rightarrow A_{dest}}$ after projection.

Case 5 consists of equality constraints where the vector $\vec{v}$ used by the left-hand side is a subset of the input tuple and the right-hand side's vector $\vec{u}$ is a subset of the output tuple. The only difference between cases 4 and 5 is the use of $\vec{v}$ and $\vec{u}$ on opposite sides of the statement.

An example of this case is the constraint on *off* as seen in DIA (see Table 4.1). Suppose DIA is the destination tensor and *off* is to be solved for: solving for *off* in the constraints will result in $off(d) = j - i$. Tuple variables j and i are known but tuple variable d is not a linear combination of known tuple variables and is bounded by ND which is also not known. In the insert abstraction, whatever constraints are present as a universal quantifier on the UF in the description are enforced. In this example, $(j-i)$ is inserted into the *off* and the constraint $\forall e1, e2 : e1 < e2 \iff off(e1) < off(e2)$ is enforced.

The order that the constraints are processed is determined by the availability of information and the RHS of the constraint and the qualities of the relation. All UFs on the RHS must either be known from the source format or have been previously processed. Our running example has 4 unknown UFs: row_m, col_m, NNZ, and P. P is processed first, after P, both row_m and col_m have relations that are functions and can be processed in any order. NNZ can be processed after either row_m or col_m. The naive implementation will be case 2. However, loop fusion and dead code elimination make it a simple assignment.

Enforce Universal Quantifiers. Any universal quantifiers present in the destination format are enforced. There are two types of universal quantifiers on an uninterpreted function: a reordering quantifier and a monotonic quantifier. Reordering universal quantifiers results in an ordering constraint placed on the entire destination tensor, while a monotonic quantifier is only applicable to the uninterpreted function being described. The Morton example below is an example of a reordering quantifier.

$$\begin{aligned}
range(row_m) &= \{0 <= i < NR\} \\
domain(row_m) &= \{0 <= x < NNZ\} \\
\forall n1, n2 : n1 < n2 &\iff MORTON(row_n(n1), col_n(n1)) \\
&\quad < MORTON(row_n(n2), col_n(n2))
\end{aligned}$$

Here the constraint on $n1, n2$ has a side effect on the order of the format. A monotonic quantifier on the other hand is local to the uninterpreted function and does not have any effect on the ordering of the tensor. An example will be *rowptr* in compressed sparse row format (CSR)- see Figure 2.6.

$$\begin{aligned}
range(rowptr) &= \{0 <= x <= NNZ\} \\
domain(rowptr) &= \{0 <= i <= NR\} \\
\forall e1, e2 : e1 < e2 &\iff rowptr(e1) <= rowptr(e2)
\end{aligned}$$

In both cases, the synthesis will ensure these constraints are satisfied. In the case of re-ordering quantifiers, the constraints will be enforced as sorting constraints. The generality of these constraints allows for user-defined functions in format specifications and this is our unique contribution. In almost all cases the code synthesized during this phase is not required for correctness and can be removed during optimization.

Generate Data Copy. The final step in the synthesis is the ”copy” code. At this point, all uninterpreted functions in the destination format have been successfully synthesized as SPF specification. The copy code copies the data from the source to the destination. The domain of the copy code is the composed relation as a set. The statement is a copy statement and the reads and writes are the source and destination data accesses respectively.

4.1.3 More Complex Example: COO to CSR

The COO to Morton COO example does not exercise the full strength of the synthesis algorithm. This section provides an overview of synthesizing the transformation code from COO to CSR. This more complex example demonstrates the importance of universal quantifiers for cases beyond ordering.

Insert Permutation & Compose. The first step adds the permutation constraints on the inverse function of

$R_{A_{CSR} \rightarrow A_{dense}}$ ensuring it is a function and then the compose operation as shown below.

$$\begin{aligned} {R_{A_{CSR} \rightarrow A_{dense}}}^{-1} = \{[i,j] \rightarrow [ii,k,jj] | ii = i \wedge col_2(k) = j \wedge j = jj \\ \wedge\, 0 \leq ii < NR \wedge rowptr(ii) <= k \wedge \\ k < rowptr(ii+1) \wedge P(i,j) = (ii,k,jj)\} \end{aligned}$$

The permutation constraint $P(i,j) = (ii,k,jj)$ can be further simplified into one dimensional permutes $P0(i,j) = ii$, $P2(i,j) = jj$, and $P1(i,j) = k$.

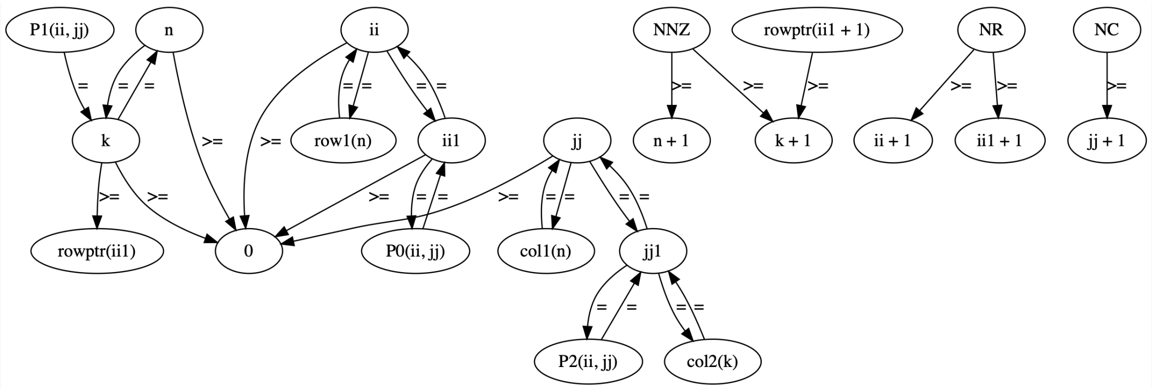


Figure 4.1: Constraint graph resulting from the composed relation $R_{A_{COO} \to A_{CSR}}$.

$$
\begin{aligned}
{R_{A_{CSR} \to A_{dense}}}^{-1} &= \{[i, j] \to [ii, k, jj] | ii = i \wedge col_2(k) = j \\
&\quad \wedge 0 \leq ii < NR \wedge rowptr(ii) <= k \wedge \\
&\quad k < rowptr(ii + 1) \wedge P0(i, j) = i \\
&\quad P1(i, j) = k\} \\
R_{A_{COO} \to A_{CSR}} &= {R_{A_{CSR} \to A_{dense}}}^{-1} \circ R_{A_{COO} \to A_{CSR}} \\
R_{A_{CSR} \to A_{CSR}} &= \{[n, ii, jj] \to [ii1, k, jj1] | ii = row_1(n) \wedge \\
&\quad jj1 = col_2(k) \wedge jj = col_1(n) \wedge 0 \leq ii < NR \wedge \\
&\quad rowptr(ii1) <= k \wedge k < rowptr(ii1 + 1) \wedge \\
&\quad P0(ii, jj) = ii1 \wedge P0(ii, jj) = jj1 \wedge jj = jj1 \\
&\quad P1(ii, jj) = k \wedge ii = ii1\}
\end{aligned}
$$

The above constraint from the composition can be represented as a constraint graph as shown in Figure 4.1. The constraint graph is internally used to represent constraints as vertices and edges. A vertex is a term in the constraint and an edge represents a relationship between two edges.

Applying Floyd Warshall's shortest path algorithm on every vertex in the constraint graph introduces a transitive closure behavior on the graph. A slight modification of Floyd-Warshal's algorithm as shown in Algorithm 1 is used to perform closure on the constraint graph.

Algorithm 1 Floyd-Warshall's modification for Transitive Closure

```
Require: $E : (c, v1, v2) : c \in \{>, <, \leq, \geq, =\} \wedge v_i \in V$
  procedure CLOSURE($G(V, E)$)                                    ▷ Constraint Graph
      for k from 1 to V do
          for i from 1 to V do
              for k from j to V do
                  cost[i][j] ← GetStrongConstraint(cost[i][k],cost[k,i],cost[i][j])
              end for
          end for
      end for
```

In the Algorithm 1, *cost* is an adjacency matrix representing the constraint graph, *GetStrongConstraint* is a function that takes in an arbitrary list of constraints and returns the strongest constraint. It assigns a number to equality(=), strong inequality $(>, <)$ and weak inequality $(\geq, \leq)$ such that `equality` > `strong inequality` > `weak inequality`. Figure 4.2 shows the new constraint graph with an explicit constrained relationships to aid synthesis.

A key step in COO to CSR is the algorithm recognizing a similar universal quantifier involving tuples n and k from source and destination format respectively. A universal quantifier on an uninterpreted function will affect the overall ordering of a tensor if and only if the tuple involves a data space tuple variable.

The data access relation for COO and CSR as shown in the table 4.2 and 4.1 is given below:

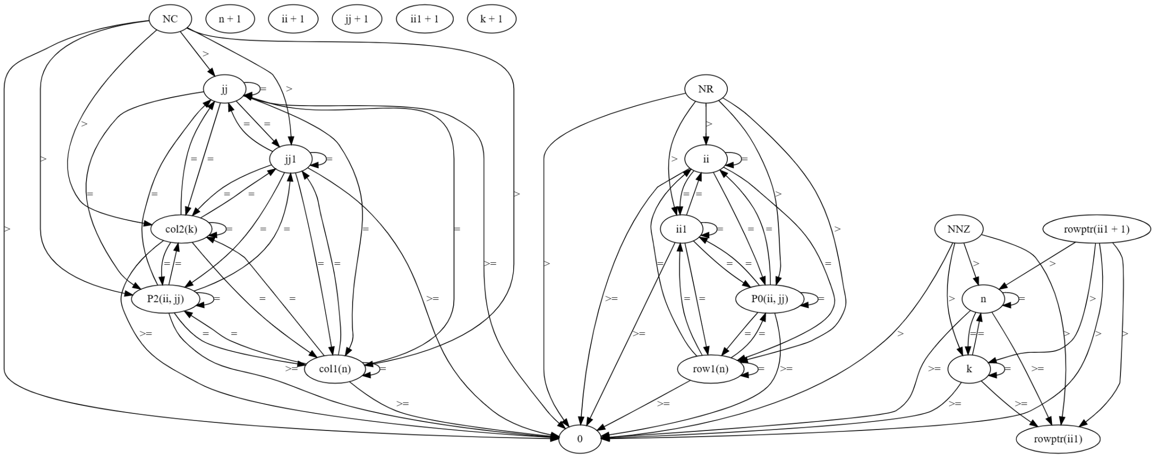


Figure 4.2: Constraint graph after closure operation on the composed relation $R_{A_{COO} \to A_{CSR}}$.

$$D_{I_{COO} \to A_{COO}} = \{[n, ii, jj] \to [n]\}$$

$$D_{I_{CSR} \to A_{CSR}} = \{[ii, k, jj] \to [k]\}$$

COO is sorted row first using the universal constraint below.

$$\forall n1, n2 : n1 < n2 \iff row(n1) <= row(n2)$$

CSR is also in a sorted form by default as shown:

$$\forall k1, k2 : k1 < k2 \iff dim0(k1) <= dim0(k2)$$

CSR does not have an uninterpreted function for row, but the synthesis algorithm is aware that $dim0$ is dense coordinate information on the first dimension. Due to the similarity in both array properties directly describing a data space tuple, reordering function $P1(i, j) = k$ can be removed in the relation and $n = k$ directly introduced.

Constraints involving permutation functions that are resolvable as a function of input tuple are also removed, this goes for $P0(i, j) = ii$ and $P2(i, j) = jj$. The constraint graph 4.2 introduces an explosive number of candidates for synthesis that could mean the same thing.

The algorithm then picks up each unknown UF and looks for constraint candidates involving such UF. Starting with *rowptr* the candidates for consideration for synthesis and its corresponding cases are shown below:

```
candidate 1: rowptr(ii1) <= n, Case: 2
candidate 2: rowptr(ii1) <= k, Case: 2
candidate 3: rowptr(ii1 + 1) >= 1, Case: 3
candidate 4: rowptr(ii1 + 1) >= n + 1, Case: 3
candidate 5: rowptr(ii1 + 1) >= k + 1, Case: 2
candidate 6: rowptr(ii1) < NNZ, Case: 2
candidate 7: -rowptr(ii1) + rowptr(ii1 + 1) - 1 >= 0, Case: none
```

The last item in the listing above does not fall into any case. The synthesis algorithm generates code for all the statements above. In CSR descriptor as seen in Table 4.1, *rowptr* has a universal quantifier as shown below:

$$\forall ii1, ii2 : ii1 < ii2 \iff rowptr(ii1) <= rowptr(ii2)$$

This quantifier does not involve a tuple used in data access so enforcing this quantifier is local to the *rowptr* UF. The statement generated for this code checks if for every $ii1 < ii2$ for the condition not been met, make at least equality condition to be met this can be generated in code as:

```
if ( not (rowptr(ii1) <= rowptr(ii2))){
   rowptr(ii2) = rowptr(ii1);
}
```

The domain of the above statement that satisfies $ii1 < ii2$ and is also valid as the domain of $rowptr$ (see Table 4.1 is given in Equation 4.1.

$$\{[ii1, ii2] : 0 <= ii1 < NR + 1 \wedge 0 <= ii2 < NR + 1 \wedge ii1 < ii2\} \tag{4.1}$$

Equation 4.2 also means the same thing and is computationally less expensive compared to Equation 4.1. In the implementation, we prefer the latter to the former.

$$\{[ii1, ii2] : 0 <= ii1 < NR + 1 \wedge ii2 = ii1 + 1\} \tag{4.2}$$

Finally, we pick up constraints from the graph 4.2 involving CSR's $col2$ as shown in the listing below

```
candidate 1: col2(n) = jj, Case: 1
candidate 2: col2(k) = jj, Case: 1
candidate 3: col2(k) = jj1, Case: 1
candidate 4: col2(n) = col1(n) , Case: 1
candidate 5: col2(k) = col1(n)  , Case: 1
candidate 6: col2(n) >= 0, Case: 3
candidate 7: col2(k) >= 0, Case: 3
candidate 8: NC > col2(n), Case: 2
candidate 9: NC > col2(k), Case: 2
```

All nine candidates in the listing are valid for code synthesis, however, one case is sufficient enough to generate code. We prefer case 1 candidates over other cases as it results in an equality constraint that fully covers the uninterpreted function. For instance, picking candidate 1 will result to an iteration space shown in Equation 4.3

$$\{[n, ii, jj] : 0 <= n < NNZ \wedge ii = row(n) \wedge jj = col(n)\} \tag{4.3}$$

A list of selected candidates for each uninterpreted function is assigned a unique execution schedule. The synthesis algorithm also works as a fixed-point solution. The algorithm picks an unknown UF and attempts to apply previously discussed operations to it, if in the case where there is no valid solution or candidates that fall into any case, the UF is placed at the back of the queue and other UFs get solved. When the algorithm has reached a point where nothing is being solved and items still remain in the queue, the program terminates early with an error indicating which uninterpreted function was not able to be solved. Fully describing computations in SPF IR also requires read and write accesses, this is computed in all selected candidates by static analysis. Figure 4.6 shows code generated from the SPF IR representation produced by the synthesis algorithm.

4.1.4 SPF Transformations for Optimization

The initial SPF representation may contain redundant or unnecessary code and use loop structures that are less than ideal for performance. We employ a collection of standard SPF transformations to improve performance.

In the synthesis process, we pick up constraints that are viable candidates for synthesizing statements for an unknown UF. However, this can produce multiple statements that do the same thing. If multiple statements cover the same data space we remove all but one of them.

There are situations where the permutation, P is not required for code correctness. This case is detected using standard dead code elimination. SPF, at its most basic, is a dataflow graph. That graph is traversed backward, starting with the live-out dataspaces. Any dataspace and corresponding computation that is not visited is removed.

Fusion combines two loops into one loop. Read reduction fusion aims at combining statements that read from the same location in memory to reduce memory footprint. Previous work [41] shows support for fusion in SPF. Multiple reads on the same data location occur in synthesis as a series of loop chains.

Producer-consumer fusion combines chains of loops that write and then read from the same data. This often results in reducing the space needed for temporary data storage. All opportunities to apply read-reduction and producer-consumer fusion are applied.

4.2 Implementation

Our implementation uses IEGenLib [51] for sparse set and relation manipulation. The synthesis algorithm generates the SPF IR using the SPF API [41] which supports code generation with polyhedral scanning. In this section we describe implementation details, algorithm complexity, and the data structures used in our work.

4.2.1 Detailed Algorithm

In this section, we will be focusing on implementation details as regards to the synthesis algorithm and its working mechanism. Algorithm 2 shows a more detailed implementation of the synthesis algorithm.

The *AddPermutation* adds a permutation UF $P([in]) = [out]$ constraint to its relation parameter. The procedure *RemovePermuteEqualToResolvableTuples* removes all unnecessary permute functions that would generate redundant code—functions that are equal to resolvable output tuples. An output tuple is termed resolvable if it can be expressed as a function of input tuple variables and/or known uninterpreted functions. The compose operation $\circ$ has been defined earlier in Section 2 Theorem 1. The procedure *BuildConstrGraph* builds a constraint graph from the constraints of a relation, an example of this graph is seen in Figure 4.1. The function *Closure* applies Floyd-Warshall's algorithm as described in Algorithm 1, this explores more possible candidates for synthesis. The procedure *GetConstraints* takes in a constraint graph and returns a list containing all the edges in the graph as constraints. All symbolic constants and uninterpreted functions are computed using the *GetSymbols* procedure in the algorithm. This is important for populating known and unknown UFs/symbolic constants; crucial for the synthesis algorithm.

Solving for a UF in an equation/constraint is important for synthesis. This ensures that the unknown UF is solved for and stays in the LHS of the equation. The *solveForUF* solves for a uf in the provided constraint. The cases earlier discussed in Section 4.1.2 are calculated by the algorithm with the function *GetSynthCase*. This function makes use of all the conditions enumerated to categorize a constraint as falling into cases 1 to 5 and a special case of undefined. Undefined cases will return false for an $IsValid$ while cases 1 to 5 will return true. The procedure *GetCaseDomain* uses projectOut (see Background Section 2) set operation to remove tuples that are not a part of the constraint c. This is an efficient way to create a domain/space that covers the constraint c from the relation $R_{src \to dest}$. The procedure *GetCaseStatement* creates a statement based on the synthesis case discussed in Section 4.1.2. The reads and writes are obtained using procedures GetReads and GetWrites respectively. This is computed by inspecting the LHS for write accesses and the RHS for read accesses in the statement generated. The *execution_id* keeps track of the execution schedules of each successfully generated component of the algorithm. Uninterpreted functions in the source and destination format are described using domain, range, and array properties. The procedure $GetUFProperty$ abstracts the call for retrieving information about a UF. If a UF has an array property and its domain does not span a data space tuple, we generate code to enforce such array property as shown in Algorithm 3.

Enforce array property procedure is detailed in Algorithm 3. The components of the $ufProperty$ include the domain, range, and array properties. Figure 4.3 shows the components of an array property used in the Algorithm specification 3. The function *BuildQuantDomain* builds a set that covers the domain for enforcing an array property. The procedure uses the LHS predicate as a constraint in the set, the quantification as the tuple declaration, and all tuple variables are restricted by the domain of the function. The *BuildQuantDomain* builds a set shown in Equation 4.4 given an array property 4.3 and domain of *rowptr* in Table 4.1.

$$\{[ii1, ii2] : 0 <= ii1 <= NR \wedge 0 <= ii2 <= NR \wedge ii1 < ii2\} \tag{4.4}$$

$$\forall \underbrace{i1, i2}_{\text{Quantifaction}} : \underbrace{ii1 < ii2}_{\text{LHS Predicate}} \iff \underbrace{rowptr(ii1) <= rowptr(ii2)}_{\text{RHS Predicate}}$$

Figure 4.3: Components of a UF Array Property

The *GetMinimalStmt* procedure returns a minimal statement required for the RHS predicate of the array property to be true. RHS predicate as shown in Figure 4.3 synthesizes a statement $rowptr(ii2) = rowptr(ii1)$. The statement generated by this step in the algorithm for the *rowptr* example is given in the listing below.

```
for(ii1 = 0; ii1 <= NR; ii1++){
    for(ii2 = ii1+1; ii2 <= NR; ii2++){
        if (not (rowptr(ii1) <= rowptr(ii2))){
            rowptr(ii2) = rowptr(ii1)
        }
    }
}
```

Generate copy code procedure creates a final copy computation from the source to the destination after all uninterpreted functions have been successfully synthesized.

4.2.2 Ordered and Unordered Set Data Structure

The abstraction in Figure 4.7 shows the data structure used in special uninterpreted functions. Special uninterpreted functions fall into cases 5 and 4. The abstraction consists of insert, comparator, sort, and get. Insert adds tuples to the abstraction, the comparator is a compile-time generated reordering function, sort reorders the inserted tuple based on the comparator and get returns the tuple for

Algorithm 2 Synthesis Algorithm

```
procedure SYNTHESIS(srcDesc, destDesc)
    $spf\_ir \leftarrow \emptyset$
    $R_{src \to dense} \leftarrow srcDesc.SparseToDenseMap$
    $R_{dest \to dense} \leftarrow destDesc.SparseToDenseMap$
    $R_{dense \to dest} \leftarrow {R_{dest \to dense}}^{-1}$
    $AddPermutation(R_{dense \to dest})$
    $RemovePermuteEqualToResolvableTuples(R_{dense \to dest})$
    $R_{src \to dest} \leftarrow R_{dense \to dest} \circ R_{src \to dense}$
    $G(V, E) \leftarrow BuildConstrGraph(R_{src \to dest})$
    $GT(V, E) \leftarrow Closure(G(V, E))$
    $C \leftarrow GetConstraints(GT(V, E))$
    $unknownUFs \leftarrow GetSymbols(R_{dense \to dest})$
    $knownUFs \leftarrow GetSymbols(R_{source \to dense})$
    $execution\_id \leftarrow 0$
    while $unknownUFs$ is not empty do
        $uf \leftarrow unknownUfs.dequeue()$
        $solvedFor \leftarrow false$
        for $c \in C$ do
            if $uf \in c$ then
                $eq \leftarrow SolveForUF(c, uf)$
                $case \leftarrow GetSynthCase(eq, R_{src \to dest})$
                if $IsValid(case)$ then
                    $domain \leftarrow GetCaseDomain(c, case, R_{src \to dest})$
                    $statement \leftarrow GetCaseStatement(c, case, R_{src \to dest})$
                    $reads \leftarrow GetReads(c, case, R_{src \to dest})$
                    $writes \leftarrow GetWrites(c, case, R_{src \to dest})$
                    $execution\_sched \leftarrow GetSchedule(execution\_id)$
                    $spf\_ir.add((statement, domain, reads, writes, execution\_sched))$
                    $execution\_id \leftarrow execution\_id + 1$
                    $solvedFor \leftarrow true$
                end if
            end if
        end for
        if $solvedFor$ is false then
            $unknownUfs.enqueue(uf)$          ▷ Place back in queue if not solved for
        else
            $knownUFs.push(uf)$
            $ufQuantifier \leftarrow GetUFQuantifier(uf)$
            if $ufQuantifier \neq \emptyset$ then
                $EnforceArrayProperty(spf\_ir, ufQuantifier, uf)$
            end if
        end if
    end while
    $GenerateCopyCode(spf\_ir, srcDesc, destDesc, execution\_id)$ return $spf\_ir$
end procedure=0
```

Algorithm 3 Enforce array property procedure

```
procedure EnforceArrayProperty($spf\_ir, ufQuantifier, uf, execution\_id$)
    ufdomain ← ufProperty.domain
    ufProperty ← $ufQuantifier.arrayProperty$
    q ← ufProperty.quantification
    lhsPred ← $GetLHSPredicate(ufProperty.predicate)$
    rhsPred ← $GetRHSPredicate(ufProperty.predicate)$
    domain ← $BuildQuantDomain(ufdomain, q, lhsPred)$
    statement ← " if (not ("+$rhsPred$+")) {"+$GetMinimalStmt(rhsPred)$+"}"
    reads ← $GetReads$(statement)
    writes ← $GetWrites$(statement)
    $spf\_ir.add((statement, domain, reads, writes, execution\_sched))$
end procedure
```

Algorithm 4 Generate copy code algorithm

```
procedure GenerateCopyCode($spf\_ir, srcDesc, destDesc, execution\_id$)
    $D_{I_{src} \to A_{src}} \leftarrow sourceDesc.DataAccess$
    $D_{I_{dest} \to A_{dest}} \leftarrow destDesc.DataAccess$
    $copyStatement \leftarrow GetCopyStatement(D_{I_{src} \to A_{src}}, D_{I_{dest} \to A_{dest}})$
    $copyDomain \leftarrow ToSet(R_{src \to dest})$
    $execution\_sched \leftarrow GetSchedule(execution\_id)$
    $spf\_ir.add((statement, domain, reads, writes, execution\_sched))$
end procedure
```

```
  // off(d)
#define off(d) off->get_inv(d)
Permute < forall e1,e2: e1<e2 <=> off(e1)< off(e2)> off = new Permute();
// off insert (j - i) used inside of some domain
off->insert( {j - i});
```

Figure 4.4: An example of code generated for Case 4 with offset in DIA

a certain position in the abstraction. Consider the constraint off$(d) = j - i$, with constraint $\forall e1, e2 : e1 < e2 \iff \text{off}(e1) < \text{off}(e2)$ code generated is shown in Figure 4.4 for insert statements and access of *off* uninterpreted function.

4.2.3 Reorder Stream on Permutation

Iterating through the source format and retrieving new positions using the get function in Figure 4.7 is expensive. In the implementation, we introduce a data structure that explores streaming. This technique iterates through the reorder function rather than the source format. This way explore keeping read accesses to the reorder function in the cache and avoid unpredictable jumps when iterating through the source format. We use a slightly modified data structure as seen in Figure 4.5. Functions insert, setComparator, sort, and getSize are similar to the initial permute data structure in Figure 4.7. The major change includes the introduction of the getMap, getDim, and a dim parameter with the constructor. The getMap function returns the new reordered position for an old position idx, and getDim returns the dense dimension for some old position idx.

4.2.4 Complexity

This section discusses the complexity of the synthesis algorithm and the resulting generated code from synthesis. The resulting code from synthesis is the inspector for

```
ReorderStream{
  ReorderStream(dim)
  void insert(Tuple tuple)
  void setComparator(Comparator comp)
  void sort()
  getSize()
  int getMap(idx)
  int getDim(dimension,idx)
};
```

Figure 4.5: Permutation Data Structure Exploring Reorder Stream

sparse format conversion that converts from one sparse format to another. It should be noted that the synthesis algorithm complexity is a compile time cost for inspector generation while the inspector complexity of the generated code is a run time cost. All evaluation done in this work in Section 4.3 evaluates the inspector code for format conversion and not the synthesis algorithm complexity.

Synthesis Algorithm Complexity

The synthesis algorithm is made up of generating the initial SPF IR and code generation from the SPF IR. The asymptotic complexity of generating the SPF IR with the synthesis algorithm is O(nm) where n is the number of unknowns and m is the number of constraints. The SPF IR supports code generation with polyhedral scanning which has an exponential complexity $O(q^p)$ where q is the number of statements in the IR and p is the tuple size of the execution schedule of the computation. In practice, the number of statements and size execution schedule is usually small and oftentimes influenced by the number of UFs present in the destination formats. The overall complexity of the synthesis algorithm all the way to code generation is given as $O(q^p)$.

Inspector Complexity

In our generated code, the complexity depends on the reorder function been present in the final generated code. In the situation where the permutation reorder function is generated, the worst case of code generated will be $O(n \log n)$ where n is the number of elements in the data of the source format. In situations when the reorder function is not generated, the worst case of the code generated will be $O(n)$. The extra cost of using the reorder function is as a result of its required sorting step.

4.3 Evaluation

We evaluated the correctness and performance of the synthesized code using a set of sparse matrices from the SuiteSparse Matrix Collection [23]. The synthesized code is serial, we do not explore parallelization opportunities. We show results for COO to CSR (COO_CSR), CSR to CSC (CSR_CSC), and COO to DIA (COO_DIA). The performance of the transformations varies depending on the target format. The COO to CSR transformation is 2.85x faster than TACO, while the more complex COO to DIA is 1.4x slower than TACO but faster than SPARSKIT and Intel MKL using a geometric average. We evaluate results for COO_MCOO by comparing our results with handwritten z-Morton step reordering in HiCOO. All speedup or slowdown comparisons use geometric averages.

4.3.1 Experimental Setup

All experiments are run on a Linux (CentOS release 7) cluster supporting 27 compute nodes, each with dual Intel Xeon E5-2680 14-core CPUs. We compile generated code and TACO code using GCC 10.2.0.

```
#define s_0(n, ii) rowptr(ii) = min(rowptr(ii),n)
#define s0(__x0, a1, __x2, a3, __x4, __x5, __x6, __x7, __x8, __x9, __x10, __x11, __x12) s_0(a1, a3);
#define s_1(n, ii) rowptr(ii + 1) = max(rowptr(ii + 1),n + 1)
#define s1(__x0, a1, __x2, a3, __x4, __x5, __x6, __x7, __x8, __x9, __x10, __x11, __x12) s_1(a1, a3);
#define s_2(e1, e2) if ( not (rowptr(e1) <= rowptr(e2))){rowptr(e2) = rowptr(e1);}
#define s2(__x0, a1, __x2, a3, __x4, __x5, __x6, __x7, __x8, __x9, __x10, __x11, __x12) s_2(a1, a3);
#define s_3(n, ii, jj, ii1, k, jj1) col2(k)=jj1
#define s3(__x0, a1, __x2, a3, __x4, a5, __x6, a7, __x8, a9, __x10, a11, __x12) s_3(a1, a3, a5, a7, a9, a11);
#define s_4(n, ii, jj, ii1, k, jj1) ACSR(ii,k,jj) = ACOO(n,ii,jj )
#define s4(__x0, a1, __x2, a3, __x4, a5, __x6, a7, __x8, a9, __x10, a11, __x12) s_4(a1, a3, a5, a7, a9, a11);
......
if (NR >= 1 && NC >= 1) {
// Case 2 : rowptr
  for(t2 = 0; t2 <= NNZ-1; t2++) {
      t4=row1_0(t1,t2);
      s0(0,t2,0,t4,0,0,0,0,0,0,0,0,0);

  }
  // Case 3: rowptr
  for(t2 = 0; t2 <= NNZ-1; t2++) {
      t4=row1_0(t1,t2);
      s1(1,t2,0,t4,0,0,0,0,0,0,0,0,0);
  }
}
// Enforcing array property on rowptr
for(t2 = 0; t2 <= NR-1; t2++) {
  s2(2,t2,0,t2+1,0,0,0,0,0,0,0,0,0);
}
if (NC >= 1 && NR >= 1) {
  // Case1: col
  for(t2 = 0; t2 <= NNZ-1; t2++) {
    t4=row1_0(t1,t2);
    t6=col1_1(t1,t2);
    s3(3,t2,0,t4,0,t6,0,t4,0,t2,0,t6,0);
  }
  // Copy code
  for(t2 = 0; t2 <= NNZ-1; t2++) {
    t4=row1_0(t1,t2);
    t6=col1_1(t1,t2);
    s4(5,t2,0,t4,0,t6,0,t4,0,t2,0,t6,0);
  }
}
```

Figure 4.6: An example of code generated COO To CSR. Code sections are omitted for clarity.

```
Permute <Comparator> {
  vector<Tuple> newPos;
  Permute(Comparator comparator)
  void sort()
  void insert(Tuple tuple)
  int get(originalPos)
}
```

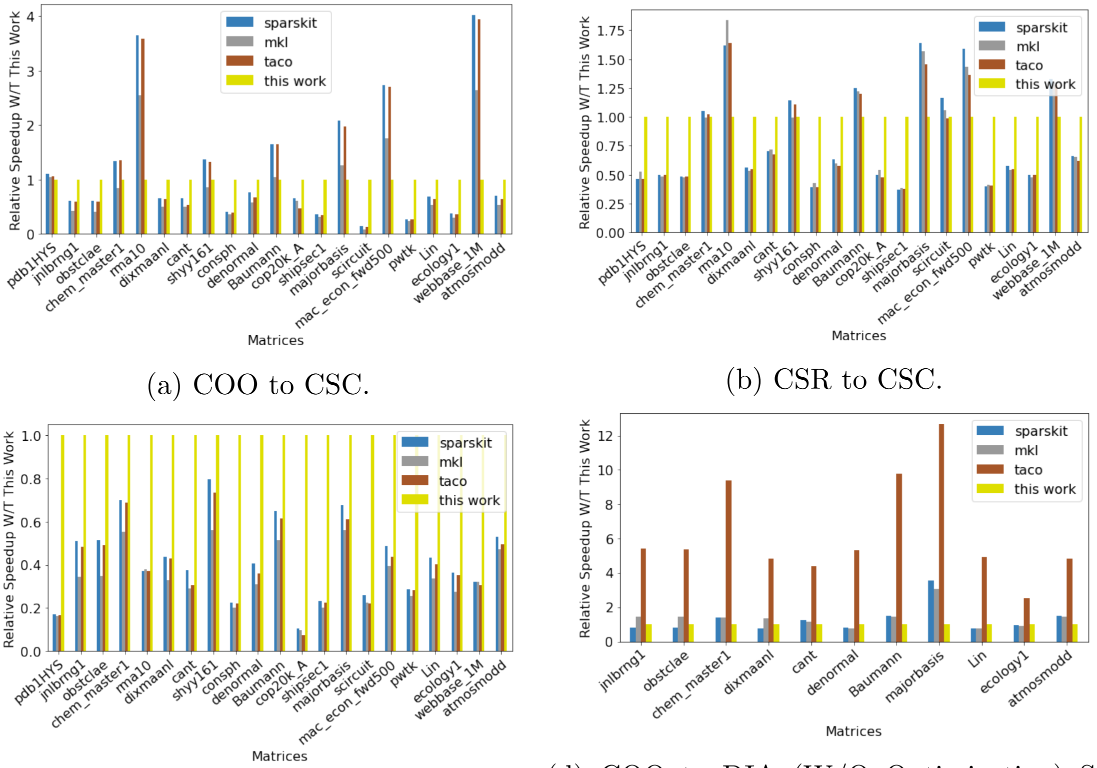
Figure 4.7: Permutation Data Structure

The performance comparison is made using the same matrix tensors used in TACO's format conversion work. Table 4.4 shows the matrices used in our evaluation. The COO matrix is assumed to be sorted lexicographically row first. Table 4.5 shows 3D tensors used in evaluating COO_MCOO reordering.

4.3.2 Performance Evaluation

We evaluate the inspector code for format conversion generated by our algorithm by comparing to Intel MKL, TACO [31], SPARSKIT [47], and hand written HiCOO's [34] z-morton ordering. Figure 4.8c shows conversion results from COO to CSR where we see a significant 2.85x speedup compared to TACO and other libraries. Code generated for COO to CSC (Figure 4.8a) and CSR to CSC (Figure 4.8b) shows a 1.3x and a 1.5x speedup on a geometric average respectively. COO to CSR shows a significant speed-up compared to CSR_CSC and COO_CSC due to the row first lexicographical ordering of the source COO format, no permute function is generated. The performance metric is the speed up of the execution times of the state of the art compared to our work (the higher the better).

We compare our COO-3D to Morton COO-3D conversion to hand-written highly optimized z-morton ordering step in Hi-COO and we also see a 1.64x slow down on a geometric average as seen in Table 4.5. Hand-written z-Morton ordering splits the



(a) COO to CSC.

(b) CSR to CSC.

(c) COO to CSR.

(d) COO to DIA (W/O Optimization) See Figure 4.9.

Figure 4.8: Performance results of generated synthesis code for COO_CSC, CSR_CSC, COO_CSR and COO_DIA. COO is assumed to be sorted lexicographically.

original tensor into smaller kernels and then applies a quick Morton sort to sort each block. This results in a significantly improved performance compared to our results, as they only sort small sections at a time. Our morton-ordered tensor conversion routine spans the whole tensors.

The overhead introduced by permutation abstraction can be amortized by parallelizing insertion and sort. Permutation of source format can also be done in place, which could potentially reduce the overhead of copying from the source to the destination format. We do not explore this currently as we assume the original source tensor will need to be available after synthesis. A faster copy from source to

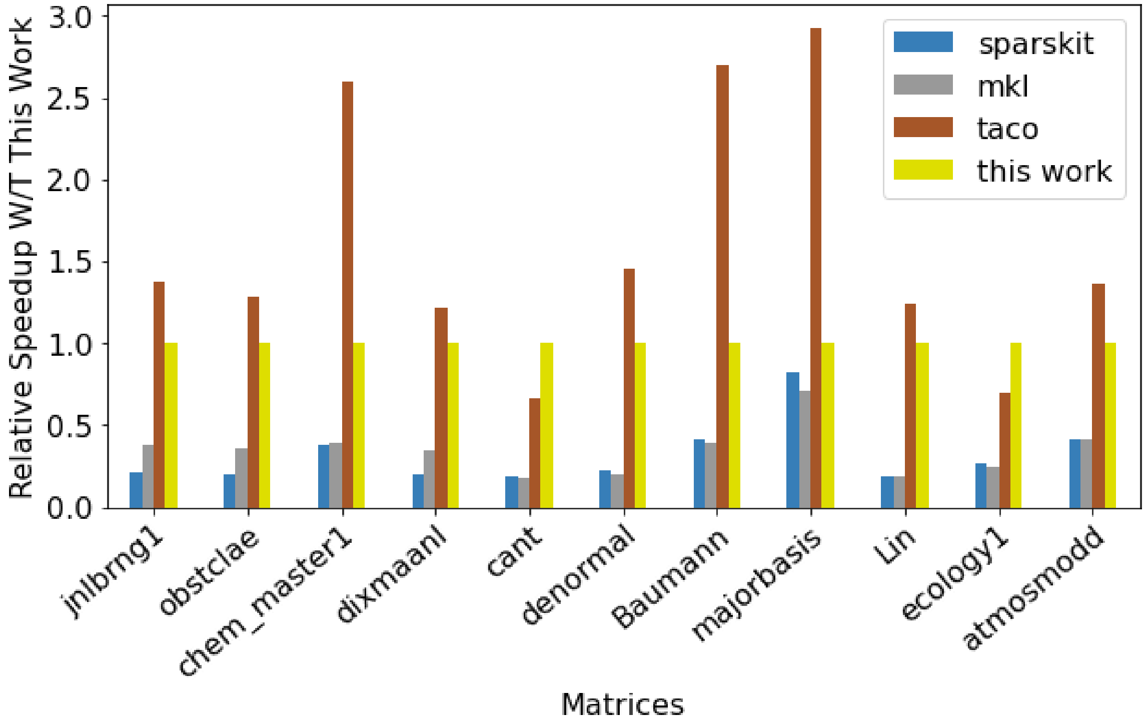


Figure 4.9: COO to DIA with binary search used to take advantage of monotonicity of synthesized offset array used in copy.

destination can be done using direct memcopy which we do not explore in this work.

Performance results for COO_DIA in Figure 4.8d show a fairly competitive performance with handwritten libraries but 5x slower on average compared to TACO. This is partially due to the fact that our optimizations cannot fuse the loops generating offset and copy code. The synthesis algorithm generates code to enforce index properties of unknown UF. The offset UF in this case has an index array property that has to be enforced before the UF is valid to be used in the copied code preventing fusion opportunities for copy code and offset code. Performance degrades with the number of diagonals. Taking a closer look at *majorbasis* (see Figure 4.8d) which showed the worst performance, the number of diagonals with nonzeros is 22 while the best performing *ecology1* has 5 diagonals. Our synthesized code tries every iteration to find the d that satisfies the constraints $off(d)+i = j$ before copying the value into the appropriate destination tensor. This constraint describes a linear search operation, which when replaced with a binary search shows a better-performing result as shown in Figure 4.9. Binary search is made possible due to the universal quantifiers on *off* (See Table 4.1). This change shows an improved result as we are 3.1x and 3.54x faster than SPARSKIT and MKL and 1.4x slower than TACO on a geometric average.

In summary, the synthesized code is competitive with the state-of-the-art, in some cases beating the performance. We anticipate that more aggressive optimization will yield better results.

Table 4.1: Format Descriptors for COO, MortonCOO (MCOO), Morton COO 3D (MCOO3)

Format	Map & Data Access	Domain Range & Array Properties
COO	$R_{A_{COO}\to A_D} = \{[n, ii, jj] \to [i, j] \mid$ $row_1(n) = i \wedge col_1(n) = j \wedge$ $ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge$ $0 \leq n < NNZ \wedge 0 \leq j < NC\}$ $D_{I_{COO}\to A_{COO}} = \{[n, ii, jj] \to [n]\}$	$range(row_1) = \{0 \leq i < NR\}$ $domain(row_1) = \{0 \leq x < NNZ\}$ $range(col_1) = \{0 \leq i < NC\}$ $domain(col_1) = \{0 \leq x < NNZ\}$
COO3D	$R_{A_{COO3D}\to A_D} = \{[n, ii, jj, kk] \to [i, j, k] \mid$ $row_1(n) = i \wedge col_1(n) = j \wedge$ $ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge$ $kk = k \wedge 0 \leq k < NZ \wedge z_1(n) = k$ $\wedge 0 \leq n < NNZ \wedge 0 \leq j < NC\}$ $D_{I_{COO3D}\to A_{COO3D}} = \{[n, ii, jj, kk] \to [n]\}$	$range(row_1) = \{0 \leq i < NR\}$ $domain(row_1) = \{0 \leq x < NNZ\}$ $range(col_1) = \{0 \leq i < NC\}$ $domain(col_1) = \{0 \leq x < NNZ\}$ $range(z_1) = \{0 \leq k < NZ\}$ $domain(z_1) = \{0 \leq x < NNZ\}$
MCOO	$R_{A_{MCOO}\to A_D} = \{[n, ii, jj] \to [i, j] \mid$ $row_m(n) = i \wedge col_m(n) = j \wedge$ $ii = i \wedge 0 \leq i < NR \wedge$ $0 \leq n < NNZ \wedge jj = j$ $\wedge 0 \leq j < NC\}$ $D_{I_{MCOO}\to A_{MCOO}} = \{[n, ii, jj] \to [n]\}$	$range(row_m) = \{0 \leq i < NR\}$ $domain(row_m) = \{0 \leq x < NNZ\}$ $range(col_m) = \{0 \leq i < NC\}$ $domain(col_m) = \{0 \leq x < NNZ\}$ $\forall n1, n2 : n1 < n2 \iff$ $MORTON(row_m(n1),$ $col_m(n1)) <$ $MORTON(row_m(n2)$ $, col_m(n2))$
MCOO3	$R_{A_{MCOO3}\to A_D} = \{[n, ii, jj, kk] \to [i, j, k] \mid$ $row_1(n) = i \wedge col_1(n) = j \wedge$ $ii = i \wedge jj = j \wedge 0 \leq j < NC \wedge$ $kk = k \wedge 0 \leq k < NZ$ $\wedge z_1(n) = k \wedge 0 \leq i < NR$ $\wedge 0 \leq n < NNZ\}$ $D_{I_{MCOO3}\to A_{MCOO3}} =$ $\{[n, ii, jj, kk] \to [n]\}$	$range(row_1) = \{0 \leq i < NR\}$ $domain(row_1) = \{0 \leq x < NNZ\}$ $range(col_1) = \{0 \leq i < NC\}$ $domain(col_1) = \{0 \leq x < NNZ\}$ $range(z_1) = \{0 \leq k < NZ\}$ $domain(z_1) = \{0 \leq x < NNZ\}$ $\forall n1, n2 : n1 < n2 \iff$ $MORTON(row_1(n1),$ $col_1(n1), z_1(n1)) <$ $MORTON(row_1(n2),$ $col_1(n2), z_1(n2))$

Table 4.2: Format Descriptors for Sorted-COO(SCOO), CSR, DIA and CSC.

Format	Map & Data Access	Domain Range & Array Properties
SCOO	$R_{A_{SCOO}\to A_D} = \{[n, ii, jj] \to [i, j] \mid$ $row_s(n) = i \wedge col_s(n) = j \wedge$ $ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge$ $0 \leq j < NC \wedge 0 \leq n < NNZ\}$ $D_{I_{SCOO}\to A_{SCOO}} = \{[n, ii, jj] \to [n]\}$	$range(row_s) = \{0 \leq i < NR\}$ $domain(row_s) = \{0 \leq x < NNZ\}$ $range(col_s) = \{0 \leq i < NC\}$ $domain(col_s) = \{0 \leq x < NNZ\}$ $\forall n1, n2 : n1 < n2 \iff$ $row_s(n1) \leq row_s(n2)$
CSR	$R_{A_{CSR}\to A_D} = \{[ii, k, jj] \to [i, j] \mid$ $ii = i \wedge jj = j \wedge col_2(k) = j$ $\wedge 0 \leq ii < NR \wedge rowptr(ii) \leq k \wedge$ $k < rowptr(ii + 1)\}$ $D_{I_{CSR}\to A_{CSR}} = \{[ii, k, jj] \to [k]\}$	$range(rowptr) = \{0 \leq n \leq NNZ\}$ $domain(rowptr) = \{0 \leq x \leq NR\}$ $range(col_2) = \{0 \leq i < NC\}$ $domain(col_2) = \{0 \leq x < NNZ\}$ $\forall ii1, ii2 : ii1 < ii2 \iff$ $rowptr(ii1) \leq rowptr(ii2)$ $\forall k1, k2 : k1 < k2 \iff$ $dim0(k1) \leq dim0(k2)$
CSC	$R_{A_{CSC}\to A_D} = \{[jj, k, ii] \to [i, j] \mid$ $\wedge 0 \leq jj < NC \wedge colptr(jj) <= k \wedge$ $k < colptr(jj + 1)\}$ $D_{I_{CSC}\to A_{CSC}} = \{[ii, k, jj] \to [k]\}$	$range(colptr) = \{0 \leq n \leq NNZ\}$ $domain(colptr) = \{0 \leq x \leq NC\}$ $range(row) = \{0 \leq i < NR\}$ $domain(row) = \{0 \leq x < NNZ\}$ $\forall jj1, jj2 : jj1 < jj2 \iff$ $colptr(jj1) \leq colptr(jj2)$ $\forall k1, k2 : k1 < k2 \iff$ $dim1(k1) <= dim1(k2)$
DIA	$R_{A_{DIA}\to A_D} = \{[ii, d, jj] \to [i, j] \mid$ $i = ii \wedge 0 \leq i < NR \wedge 0 \leq d < ND$ $\wedge j = i + off(d) \wedge 0 <= j < NC\}$ $D_{I_{DIA}\to A_{DIA}} = \{[ii, d, jj] \to [kd] \mid\}$ $kd = ND * ii + d\}$	$domain(off) = \{0 \leq x \leq ND\}$ $\forall d1, d2 : d1 < d2 \iff$ $off(d1) < off(d2)$

Table 4.3: this table are the unknown uninterpreted functions (UFs) for the running example $COO \rightarrow COO_M$. Under each are the constraints related to that UF.

row_m	col_m	NNZ	P
$row_1(n1) = row_m(n2)$ $ii = row_m(n2)$ $\forall n2, n2' : n2 < n2' \iff MORTON(ii, jj) < MORTON(ii, jj)$	$col_1(n1) = col_m(n2)$ $jj = col_m(n2)$ $\forall n2, n2' : n2 < n2' \iff MORTON(ii, jj) < MORTON(ii, jj)$	$domain(row_m) = \{0 <= x < NNZ\}$	$P(ii, jj) = [n2, ii, jj]$ $\forall n2, n2' : n2 < n2' \iff MORTON(ii, jj) < MORTON(ii, jj)$

Table 4.4: Matrices statistics used in evaluating COO_CSR, CSR_CSC,COO_DIA.

Matrix	Dimensions	NNZ
pdb1HYS	36.4K × 36.4K	4.3M
jnlbrng1	40.0K × 40.0K	199K
obstclae	40.0K × 40.0K	199K
chem_master1	40.4K × 40.4K	201K
rma10	46.8K × 46.8K	2.4M
dixmaanl	60.0K × 60.0K	300K
cant	62.5K × 62.5K	4.0M
shyy161	76.5K × 76.5K	330K
consph	83.3K × 83.3K	6.0M
denormal	89.4K × 89.4K	1.2M
Baumann	112K × 112K	748K
cop20k_A	121K × 121K	2.6M
shipsec1	141K × 141K	3.6M
majorbasis	160K × 160K	1.8M
scircuit	171K × 171K	959K
mac_econ_fwd500	207K × 207K	1.3M
pwtk	218K × 218K	11.5M
Lin	256K × 256K	1.8M
ecology1	1.00M × 1.00M	5.0M
webbase1M	1.00M × 1.00M	3.1M
atmosmodd	1.27M × 1.27M	8.8M

Table 4.5: Tensors used in evaluating COO3D_MCOO3.

Tensor	Dim	Mode	NNZ	Exec Time(s)	
				M-Hicoo	Ours
darpa	22K × 22K × 24M	3	28M	11.85	20.13
f-m	23M × 23M × 166	3	100M	49.35	78.24
fb-s	39M × 39M × 532	3	140M	70.52	114.45

Table 4.6: Automatic sparse format conversion support in our work compared to others.

Format Description Support			
Tool	Mapping	Re-order	Universal Quantifiers
TACO [31]	✓	×	×
Nandy et. al [39]	×	✓	✓
Venkat et. al [54]	×	✓	✓
This work	✓	✓	✓

CHAPTER 5

RELATED WORK

This chapter discusses prior work most closely related to this work with a focus on polyhedral model tools, program dependence graphs, sparse polyhedral model tools, sparse tensor re-ordering, automatic sparse layout conversion (see Table 4.6), handwritten sparse layout conversion, and program synthesis.

5.1 Polyhedral Model Tools

Tools such as Polly [26], Pluto [12], Loopy [38], PolyMage [37] use the polyhedral model to transform regular codes. Polly automatically detects and transforms important code sections in the LLVM IR, breaking the limitations of most tools limited to a single source language. PolyMage is a domain-specific language that automates the generation of efficient implementations of image processing pipelines. Halide [44] is another compiler for generating code for image computing algorithms. Halide separates algorithm and scheduling specification, thereby allowing optimization engineers to write different schedules for optimum performance. Their work also uses autotuning to generate efficient code by performing a stochastic search to find good schedules for the algorithm.

Pluto [12] is a fully automatic source-to-source transformation tool that optimizes programs for parallelism and locality. Pluto uses integer linear programming to decide on optimal code using parallelism and locality as part of its cost functions. Loopy, is a tool that allows a programmer to describe loop transformation at high level and verifies the transformation for correctness. Loopy, like Polly is implemented in LLVM. *isl* [56] is a tool for manipulating sets and relations in the polyhedral model. This tool forms the basis for affine transformations used in all the tools earlier discussed in this section [12, 37, 38, 44].

5.2 Program Dependence Graphs

PDFGs are based on a line of research started by Ferrante et. al, [25] with their work on program dependence graphs. Existing work demonstrates the benefit of polyhedral dataflow optimizations. Olschanowsky et al. demonstrated this benefit on a computational fluid dynamic benchmark [40]. Davis et al. automated the experiments from the previous work using modified macro dataflow graphs [22]. The Concurrent Collections (CnC) programming model [14] is a dataflow and stream-processing language where a program is a graph of computation nodes that communicate with each other. DFGR [48] is based on CnC and Habanero-C [7] programming models and allows developers to express programs at a high level with dataflow graphs as an intermediate representation. Our work uses the dataflow graph to focus on serial code optimization while DFGR and CnC explores parallelism. Stateful dataflow multigraphs (SDFGs) [8] are a data-centric intermediate representation that enables separating code definition from its optimization. Our work differ from SDFGs due to the use of the polyhedral model. The graphs are not the intermediate representation, but a view of that representation. Any graph operation performed to transform the graph is translated to relations and applied to the underlying polyhedral representation.

5.3 Sparse Polyhedral Model Tools

Work done on representing indirect memory accesses in a computation using the polyhedral model has seen the development of tools such as Omega [30], and Chill [45]. Omega [30] is a C++ library for manipulating integer tuple relations and sets. Codegen+ [18] is built on omega and generates code with polyhedral scanning in the presence of uninterpreted functions. Chill [45] is a polyhedral compiler transformation and code generation framework that uses Codegen+ for code generation. It allows users to specify transformation sequences through scripts. Our work differs from this work as we represent a holistic view of a computation and we support more precise transformations in the presence of sparse computations.

5.4 Sparse Tensor Reordering

Sparse tensor reordering involves changing the order of non-zero entries in sparse formats to improve spatial or temporal locality. This includes heuristic techniques: BFS-MCS a breadth-first search over maximum cardinal search family, and Lexi-Order an extension of the doubly lexical ordering of matrices to tensors [35]. Another approach to tensor reordering is to use discrete cosine transform (DCT) to compress Convolutional Neural Networks (CNN) [35]. Our work is similar to this class of work as we introduce a formal approach to specify reordering functions for the automatic synthesis of conversion routines.

5.5 Automatic Sparse Layout Conversion

Table 4.6 shows work on automatic data layout transformations. Mapping includes work that uses a function to describe the relationship between the sparse space and dense space, reordering is a class of work with data layout shuffling, and universal quantifiers describe array properties and integrate such information in dependence analysis and optimizations.

Script-based techniques introduce a set of transformations when combined in certain order to facilitate data layout transformations [39, 53, 54]. Compiler transformations are used as building blocks to write scripts that transform from one data format to another. This approach requires 2^n scripts to be manually written; one for each possible combination of formats. Our work differs from this work as we focus on format conversion, and use a descriptor-based approach using maps to automatically synthesize code between formats. Our approach is similar as we build on the sparse polyhedral framework; we also both support reordering and make use of universal quantifiers which opens up opportunities for dependence tests and optimizations.

Arnold et al. [6] uses a functional little language (LL) to describe sparse layout through its conversion process from a dense matrix. LL enables generating and proving the correctness of sparse computation under different layouts. However, loop transformations like fusion and tiling are not supported.

TACO [19, 20, 31] is a tensor algebra compiler that defines sparse layouts using a set of names for each dimension of the tensor called level formats. Level functions are defined for this format to support primitive operations of the dimension, including iterating, accessing, and assembling. Format conversion is achieved in TACO by mapping to and from the dense space, analyzing the tensor's structural statistics, and assembling the destination layout using the level functions. This work, however, does not consider attributes such as universal quantifiers and reordering as shown in Table 4.6.

Bik et al. [10] sparse tensor work is complementary to our work, they describe sparse formats with level properties and generate sparse computation code in MLIR. In a case where there is the need to transform from one format to another, level properties can be translated to our high-level sparse description after which our synthesis algorithm is applied. Our synthesis produces a high-level intermediate representation that can be extended to generate code for MLIR.

5.6 Optimal Tensor Layout

Bik et al. [11] and SIPR [43] optimize computation involving sparse tensors by suggesting more efficient layouts from statically analyzing the computation. Sparso [46] optimizes a sequence of tensor computation using context-driven collective reordering analysis and matrix property discovery. Sparso can determine automatically based on static and runtime information when should layout be converted using pre-defined library routines. However, these works do not target the conversion routine: either excluding them from consideration or treating them as a black box.

5.7 Manual Sparse Format Conversion

Sparskit provides various functionalities for dealing with sparse matrices. It helps to translate one matrix form to others [47]. It supports 12 different storage formats for matrices. Intel MKL is another library that provides various routines and functionalities to perform computations on sparse matrices [58]. Sometimes to get to a destination format, an intermediary format has to be converted first. Our approach is different from this approach as we require n description to automatically synthesize 2^n conversion routines.

5.8 Program Synthesis

Satisfiability modulo theories (SMT) solvers and higher-order logic (HOL) provers are commonly used to check if the generated code agrees with the provided specification. Synthesizing a general program would result in an unmanageable search space of the generated code, thus constraints and heuristics are provided.

Sketching [13, 49, 50] is an approach to program synthesis that limits the scope of the synthesis to low-level details in an algorithm sketch or meta-sketches provided by the programmer. Syntax-guided synthesis [4] uses a counter-example-guided-inductive-synthesis strategy for solving the synthesis problem under valid programs following a set of syntax. More recently, Knoth et al. [32] introduced a type system that provides automatic amortized resource analysis to use as a heuristic during the synthesis process.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter gives an overal summary of the contributions of this dissertation and discusses directions for future work.

6.1 SPF Intermediate Representation

This work presents an object-oriented API to the sparse polyhedral library. The API provides a standard interface to fully specify a computation in the Sparse Polyhedral framework from a single entry point. The single entry point is enabled by tight integration among IEGenLib, CodeGen+, and PDFGs. PDFGs have been expanded to represent non-affine loop boundaries, imperfect loop nests, and loop carried dependences. The API does not currently support early exits, loops where the bounds are modified within the loop nests, and loops without bounds such as while loops. In the future, we hope to explore techniques such as those from [9] to overcome the limitations of representing unbounded loops and exit predicates.

We provide support for combining Computations through inlining. This increases reusability and reliability. The code generated by a computation encompasses statement macros and, when needed, variable declarations. This means that in the future, memory layout decisions for temporary storage can be made entirely within the SPF.

Future tools will use this API to generate the SPF IR, manipulate it, and produce optimized code from legacy applications and custom high-level or domain-specific languages. Additionally, this API will be used when iteratively transforming a computation; the PDFG will be displayed at every step to guide the performance expert's decisions.

6.2 Code Synthesis

In this work, we introduce an approach to formally describe sparse tensor formats and synthesize translation code between them using the sparse polyhedral framework. This work presents a formal definition of sparse tensor formats and an automated approach to synthesize the transformation between formats. This approach is unique in that it supports ordering constraints not supported by other approaches and synthesizes the transformation code in a high-level intermediate representation suitable for applying composable transformations such as loop fusion and temporary storage reduction. We demonstrate that the synthesized code for COO to CSR with optimizations is 2.85x faster than TACO, Intel MKL, and SPARSKIT while the more complex COO to DIA is 1.4x slower than TACO but faster than SPARSKIT and Intel MKL using the geometric average of execution time.

6.3 Future Directions

For future work, we intend to explore more transformations in SPF to improve the overall performance of our work. Automatically guiding users in selecting the best sparse formats would be an interesting direction. Incorporating data layout transformations as part of a preexisting compiler pipeline is a direction we will explore.

In the synthesis work, there is no formal way to verify the validity of sparse descriptors. Sparse descriptors not properly described in our work cause a halting problem in our algorithm. All sparse descriptors defined in this work have been tested to work and generate correct code empirically. An interesting future work will be to formally verify the validity of a sparse format descriptor. Exploring support for abstract data type transformation is a possible direction for future work. A great direction for this work is full integration with pre-existing compilers such as MLIR, LLVM, and GCC. Integrating with compilers will mean data layouts can be automatically transformed for optimal performance.

REFERENCES

[1] gnu compiler collection (gcc) internals: gimple.

[2] IEGenLib library, sc16 artifact github repository. https://github.com/CompOpt4Apps/IEGenLib/tree/SC16_IEGenLib, 2019.

[3] Khalid Ahmad, Hari Sundar, and Mary Hall. Data-driven mixed precision sparse matrix vector multiplication for gpus. *ACM Trans. Archit. Code Optim.*, 16(4), December 2019.

[4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[5] Aaron Orenstein Anna Rift, Tobi Popoola. Sparse polyhedral framework inpector/executor spf-ie, 2020.

[6] Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. Specifying and verifying sparse matrix codes. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 249–260, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 735–736, New York, NY, USA, 2009. Association for Computing Machinery.

[8] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage*

and Analysis, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, volume LNCS 6011, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Aart J.C. Bik. Compiler support for sparse tensor computations in mlir, 2021. https://youtu.be/x-nHc3hBxHM.

[11] AartJ.C. Bik and HarryA.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 57–75. Springer Berlin Heidelberg, 1994.

[12] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15, 2008.

[13] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 775–788, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Concurrent collections. *Sci. Program.*, 18(3–4):203–217, August 2010.

[15] Chun Chen. Polyhedra scanning revisited. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 499–508, 2012.

[16] Chun Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508, New York, NY, USA, 2012. ACM.

[17] Chun Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508, June 2012.

[18] Chun Chen. Polyhedra scanning revisited. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[19] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018.

[20] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 823–838, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 265â277, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Eddie C Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 265–277. ACM, 2018.

[23] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[24] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[25] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[26] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT '11)*, page None, 2011.

[27] Mary Hall, Jacqueline Chame, Jaewook Shin, Chun Chen, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[28] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. ALTO: Adaptive linearized storage of sparse tensors. *Proceedings of the International Conference on Supercomputing*, pages 404–416, 2021.

[29] Maxime R. Hugues and Serge G. Petiton. Sparse matrix formats evaluation and optimization on a GPU. *Proceedings - 2010 12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010*, pages 122–129, 2010.

[30] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, University of Maryland at College Park, March 1995.

[31] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[32] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 253–268, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Chris Lattner and Vikram Adve. The LLVM Instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, ...*, pages 1–20, 2002.

[34] Jiajia Li, Jimeng Sun, and Richard Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 19:1–19:15, Piscataway, NJ, USA, 2018. IEEE Press.

[35] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 227–237, New York, NY, USA, 2019. ACM.

[36] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. Extending index-array properties for data dependence analysis. In *Languages and Compilers for Parallel Computing (LCPC)*, 2018.

[37] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage. *ACM SIGPLAN Notices*, 50(4):429–443, mar 2015.

[38] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9837 LNCS(July):383–402, 2016.

[39] Payal Nandy, Mary Hall, Eddie C. Davis, Catherine Olschanowsky, Mahdi Soltan Mohammadi, Wei He, and Michelle Mills Strout. Abstractions for specifying sparse matrix data transformations. In *The 8th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 23 2018.

[40] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 793–804, 3 Park Ave, New York, NY, USA, 2014. IEEE Press, IEEE Press.

[41] Tobi Popoola, Ravi Shankar, Anna Rift, Shivani Singh, Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. An object-oriented interface to the sparse polyhedral library. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1825–1831, 2021.

[42] Tobi Popoola, Tuowen Zhao, Aaron St. George, Kalyan Bhetwal, Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. Reproduction package for code synthesis for sparse tensor format conversion and optimization. March 2023.

[43] William Pugh and Tatiana Shpeisman. Sipr: A new framework for generating efficient code for sparse matrix computations. In *Proceedings of the Eleventh International Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, North Carolina, August 1998.

[44] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Frédo Durand, Connelly Barnes, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–530, 2013.

[45] Mary Hall research group. Chill - the composable high-level loop source-to-source translator. https://github.com/CtopCsUtahEdu/chill, 2019.

[46] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. Sparso: Context-driven optimizations of sparse linear algebra. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 247–259, 2016.

[47] Yousef Saad. Sparskit: a basic tool kit for sparse matrix computations, 1994.

[48] Alina Sbirlea, Louis-Noel Pouchet, and Vivek Sarkar. Dfgr an intermediate graph representation for macro-dataflow programs. In *2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, pages 38–45, 2014.

[49] Armando Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, page 4–13, Berlin, Heidelberg, 2009. Springer-Verlag.

[50] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 404–415, New York, NY, USA, 2006. Association for Computing Machinery.

[51] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the Sparse Polyhedral Framework. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7760 LNCS:61–75, 2013.

[52] Michelle Mills Strout, Geri George, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2012.

[53] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. *SIGPLAN Not.*, 50(6):521–532, jun 2015.

[54] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, 2014.

[55] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[56] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Lecture Notes in Computer Science,*, pages 299–302. Springer, September 2010.

[57] Kelly W, Maslov V, Pugh W, Rosser E, Shpeisman T, and Wonnacott D. The Omega Library interface guide. *University of Maryland at College Park Mar 1995*, 1995.

[58] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

6.4 Synthesis Artifact Appendix

6.4.1 Abstract

This artifact introduces a technique for data layout transformations based on constrained relationships between different forms of data. In this artifact, we apply this technique to generate code for transforming from one source format to another. We provide a docker container to replicate results. To ease testing we provide already generated transformation codes wrapped around necessary macros to evaluate our work. We also provide artifacts from the state-of-the-art discussed in our work.

6.4.2 Artifact check-list (meta-information)

- **Algorithm: Constraint solving, polyhedral code generation, optimizations**
- **Compilation: Known to compile with GCC v10**
- **Transformations: fusion, dead-code elimination**
- **Data set: SuiteSparse Matrix Collection [24] see Table 6.1**
- **Metrics: Relative speedup**
- **Output: Graph plots in paper**

- **How much disk space required (approximately)?: 35 Gig Docker Image Space**
- **How much time is needed to prepare workflow (approximately)?: 20 mins to download artifact files**
- **How much time is needed to complete experiments (approximately)?: 2 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Archived (provide DOI)?:Yes, at the following link: https://doi.org/10.1145/3554347** [42]

6.4.3 Description

How to access

Artifact material can be publicly accessed using this link https://doi.org/10.1145/3554347 [42]. The artifact contains a README.MD, License.txt, source codes, and scripts for replicating results in this paper.

Hardware dependencies

Known to work on amd64 and intel architecture.

Software dependencies

Docker Container

Matrix	Dimensions	NNZ
pdb1HYS	36.4K × 36.4K	4.3M
jnlbrng1	40.0K × 40.0K	199K
obstclae	40.0K × 40.0K	199K
chem_master1	40.4K × 40.4K	201K
rma10	46.8K × 46.8K	2.4M
dixmaanl	60.0K × 60.0K	300K
cant	62.5K × 62.5K	4.0M
shyy161	76.5K × 76.5K	330K
consph	83.3K × 83.3K	6.0M
denormal	89.4K × 89.4K	1.2M
Baumann	112K × 112K	748K
cop20k_A	121K × 121K	2.6M
shipsec1	141K × 141K	3.6M
majorbasis	160K × 160K	1.8M
scircuit	171K × 171K	959K
mac_econ_fwd500	207K × 207K	1.3M
pwtk	218K × 218K	11.5M
Lin	256K × 256K	1.8M
ecology1	1.00M × 1.00M	5.0M
webbase1M	1.00M × 1.00M	3.1M
atmosmodd	1.27M × 1.27M	8.8M

Table 6.1: Matrices statistics used in evaluating COO_CSR, CSR_CSC, COO_CSC

Data sets

Table 6.1 shows data used in evaluating COO_CSR, CSR_CSC, COO_CSC, and COO_DIA can be found in the docker container obtained from SuiteSparse Matrix Collection [24]. Table 6.2 shows tensors used for Morton re-ordering evaluation.

6.4.4 Installation

Installation instructions can be found in the README.MD document in the root folder of the artifact.

Tensors	Dimensions	Mode	NNZ
darpa	22K × 22K × 24M	3	28M
fb-m	23M × 23M × 166	3	100M
fb-s	39M × 39M × 532	3	140M

Table 6.2: Tensors used in evaluating COO3D_MCOO3

6.4.5 Experiment workflow

There are five experiments conducted in this work, we have provided scripts that will run both the state-of-the-art, and our work for each conversion routine. We also provide scripts to generate the figures in the paper. Kernels generated by our code generation tool have already been optimized and added to a driver, this is also the case for Taco's format conversion tool. We do provide a convenient script (run.sh) to run all experiments to replicate results in the paper. Instructions for the experiment workflow can be found in the README.MD. The experiment will generate all 4 graphs from the paper.

1. coocsr.png

2. csrcsc.png

3. coodia.png

4. coocsc.png

6.4.6 Evaluation and expected results

Results should be consistent with the figures in the paper. While there might be some slight changes in the result.

6.4.7 Notes

For some tests in COO_DIA, expect bad allocation due to the 75 percent zeros introduced when converting from these matrices to DIA. In the paper, we do not include these tensors in the result sections and will be filtered out before graphs are generated.