Code Synthesis for Sparse Tensor Format Conversion and Optimization

Tobi Popoola
tobipopoola@u.boisestate.edu
Computer Science
Boise State University
USA

Tuowen Zhao
Computer Science
University of Utah
USA
ztuowen@gmail.com

Aaron St. George
Kalyan Bhetwal
aaron.george@u.boisestate.edu
kalyanbhetwal@u.boisestate.edu
Computer Science
Boise State University
USA

Michelle Mills Strout
Computer Science
University of Arizona
USA
mstrout@cs.arizona.edu

Mary Hall
School of Computing
University of Utah
USA
mhall@cs.utah.edu

Catherine Olschanowsky
Computer Science
Boise State University
USA
catherineolschan@boisestate.edu

Abstract

Many scientific applications compute using sparse data and store that data in a variety of sparse formats because each format has unique space and performance benefits. Optimizing applications that use sparse data involves translating the sparse data into the chosen format and transforming the computation to iterate over that format. This paper 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.

Keywords: Sparse Format Synthesis, Re-ordering

1 Introduction

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 paper presents an approach that describes sparse tensor formats and synthesizes translation code among them.

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.

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 [11] and HICOO [17]. 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 [1, 30].

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

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

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. Based on the polyhedral framework, SPF uses a mathematical infrastructure based on sets and relations to express and transform computations. SPF builds on previous efforts including Omega [13] by using uninterpreted functions to abstract non-affine constraints on execution schedules. 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 [21]. 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.

The contributions of this paper 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 shows that our approach is competitive or outperforms TACO in cases where a comparison was possible.

2 Background

In this section, we discuss the necessary background for our work. Sparse polyhedral model, SPF internal representation and sparse formats are fundamental concepts used in this work.

2.1 Sparse Polyhedral Model

The polyhedral model is a mathematical representation of execution schedules used for loop transformations and dependence analysis. It represents computations as sets and relations. Iteration spaces are represented with sets and data dependences are represented using relations. Combined iteration spaces and data dependences provide a partial ordering for the target computation. Within the partial ordering, the order of execution can be altered by applying relations to the iteration space. These relations are referred to as transformations.

Consider the following example of an affine loop.

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

The iteration space, expressed in the polyhedral model is a set with all valid combinations of the tuple $[i, j]$.

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

One can apply the following relation to the set to do loop interchange.

$$IC = \{[i, j] \rightarrow [j, i]\}$$
$$I' = IC(I)$$

Performing code generation on I' yields the following code.

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

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+ [6] built on Omega [14] and precise set and relation operations in the presence of uninterpreted functions with IEGenLib [27]. Uninterpreted functions (UF) are a special case of symbolic constants. Uninterpreted functions represent data structures such as index arrays in sparse data formats. In sparse computations, array accesses become part of the loop bounds in computation, this cannot be modeled in the polyhedral framework.

The example below prints the coordinates of nonzeros in a Compressed Sparse Row (CSR) format.

```
for(int i=0; i<N; i++){
  for(int k=rowptr[i]; k <  rowptr[i+1] ;k++){
     int j  = col[k];
     printf("i: %d, j:%d\n",i,j);
  }
}
```

The iteration space has constraints involving uninterpreted functions (*rowptr* and *col*).

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

SPF provides mechanisms to further describe uninterpreted functions. This information is used for data dependence optimizations [19] and in this work, for code synthesis. Universal quantifiers are used to declare index array properties such as monotonicity.

The sparse polyhedral model employs the inspector/executor paradigm to enable the use of run-time information

Figure 1. Sparse Matrix Formats.

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. Inspectors can be reasoned as the population of uninterpreted functions and executors can be reasoned to use these uninterpreted functions to guide transformations.

2.2 SPF - Internal Representation

The SPF-IR [21] formalizes the sparse polyhedral model as an intermediate representation and provides a low-level API for sparse transformations and code generation. The SPF-IR API integrates CHiLL [7], CodeGen+ [6], Omega [13], and IEGenLib [27]. It provides a computation class to express a computation or series of computations. For one specific computation, we need to define data spaces, statements, data dependences, and execution schedules. Once constructed, a code generation algorithm based on Fourier Motzkin elimination can generate C code or a visual data flow graph to help performance engineers identify optimization opportunities.

The example below shows the SPF representation that corresponds to the CSR example above.

```
Computation* comp = new Computation();
Stmt* sps0 = new Stmt(
"printf(\"i: %d, j:%d\n\",i,j);",
"{[i,k,j]: 0 <= i < N && rowptr(i) <= k <
rowptr(i+1) && j = col(k)}","{[0]}", {},{});
comp->addStmt(sps0);
string code = comp->codeGen();
```

This example shows only a single statement in the Computation. Before the *codegen* function is invoked transformations can be applied through the API to transform the execution schedule.

2.3 Sparse Formats

Sparse formats describe how sparse coordinates and corresponding data are stored and are often based on sparse matrix formats. Figure 1 shows a few of the most common sparse matrix formats including the coordinate format. 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 [17] and Alto [12]. These formats are more complex and involve sorting and other data structures.

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.

3 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, 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.

3.1 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.

Format	Map & Data Access	Domain/Range	Universal Quantifiers
COO	$R_{A_{COO}\rightarrow A_D} = \{[n, ii, jj] \rightarrow [i, j] \mid row_1(n) = i \wedge col_1(n) = j \wedge ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge 0 \leq j < NC \wedge 0 \leq n < NNZ\}$ $D_{I_{COO}\rightarrow A_{COO}} = \{[n, ii, jj] \rightarrow [n]\}$	$range(row_1) = \{0 \leq i < NR\}$ $domain(row_1) = \{0 \leq x < NNZ\}$ $range(col_1) = \{0 \leq i < NC\}$ $domain(col_1) = \{0 \leq x < NNZ\}$	
COO3D	$R_{A_{COO3D}\rightarrow A_D} = \{[n, ii, jj, kk] \rightarrow [i, j, k] \mid row_1(n) = i \wedge col_1(n) = j \wedge z_1(n) = k \wedge ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge 0 \leq j < NC \wedge kk = k \wedge 0 \leq k < NZ \wedge 0 \leq n < NNZ\}$ $D_{I_{COO3D}\rightarrow A_{COO3D}} = \{[n, ii, jj, kk] \rightarrow [n]\}$	$range(row_1) = \{0 \leq i < NR\}$ $domain(row_1) = \{0 \leq x < NNZ\}$ $range(col_1) = \{0 \leq i < NC\}$ $domain(col_1) = \{0 \leq x < NNZ\}$ $range(z_1) = \{0 \leq k < NZ\}$ $domain(z_1) = \{0 \leq x < NNZ\}$	
MCOO	$R_{A_{MCOO}\rightarrow A_D} = \{[n, ii, jj] \rightarrow [i, j] \mid row_m(n) = i \wedge col_m(n) = j \wedge ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge 0 \leq j < NC \wedge 0 \leq n < NNZ\}$ $D_{I_{MCOO}\rightarrow A_{MCOO}} = \{[n, ii, jj] \rightarrow [n]\}$	$range(row_m) = \{0 \leq i < NR\}$ $domain(row_m) = \{0 \leq x < NNZ\}$ $range(col_m) = \{0 \leq i < NC\}$ $domain(col_m) = \{0 \leq x < NNZ\}$	$\forall n1, n2 : n1 < n2 \iff MORTON(row_m(n1), col_m(n1)) < MORTON(row_m(n2), col_m(n2))$
MCOO3	$R_{A_{MCOO3}\rightarrow A_D} = \{[n, ii, jj, kk] \rightarrow [i, j, k] \mid row_1(n) = i \wedge col_1(n) = j \wedge z_1(n) = k \wedge ii = i \wedge jj = j \wedge 0 \leq i < NR \wedge 0 \leq j < NC \wedge kk = k \wedge 0 \leq k < NZ \wedge 0 \leq n < NNZ\}$ $D_{I_{MCOO3}\rightarrow A_{MCOO3}} = \{[n, ii, jj, kk] \rightarrow [n]\}$	$range(row_1) = \{0 \leq i < NR\}$ $domain(row_1) = \{0 \leq x < NNZ\}$ $range(col_1) = \{0 \leq i < NC\}$ $domain(col_1) = \{0 \leq x < NNZ\}$ $range(z_1) = \{0 \leq k < NZ\}$ $domain(z_1) = \{0 \leq x < NNZ\}$	$\forall n1, n2 : n1 < n2 \iff MORTON(row_1(n1), col_1(n1), z_1(n1)) < MORTON(row_1(n2), col_1(n2), z_1(n2))$
CSR	$R_{A_{CSR}\rightarrow A_D} = \{[ii, k, jj] \rightarrow [i, j] \mid ii = i \wedge jj = j \wedge col_2(k) = j \wedge 0 \leq ii < NR \wedge rowptr(ii) \leq k \wedge k < rowptr(ii + 1)\}$ $D_{I_{CSR}\rightarrow A_{CSR}} = \{[ii, k, jj] \rightarrow [k]\}$	$range(rowptr) = \{0 \leq n \leq NNZ\}$ $domain(rowptr) = \{0 \leq x \leq NR\}$ $range(col_2) = \{0 \leq i < NC\}$ $domain(col_2) = \{0 \leq x < NNZ\}$	$\forall ii1, ii2 : ii1 < ii2 \iff rowptr(ii1) \leq rowptr(ii2)$ $\forall k1, k2 : k1 < k2 \iff ii * NR + col_2(k1) < ii * NR + col_2(k2)$
CSC	$R_{A_{CSC}\rightarrow A_D} = \{[jj, k, ii] \rightarrow [i, j] \mid 0 \leq jj < NC \wedge colptr(jj) <= k \wedge k < colptr(jj + 1)\}$ $D_{I_{CSC}\rightarrow A_{CSC}} = \{[ii, k, jj] \rightarrow [k]\}$	$range(colptr) = \{0 \leq n \leq NNZ\}$ $domain(colptr) = \{0 \leq x \leq NC\}$ $range(row) = \{0 \leq i < NR\}$ $domain(row) = \{0 \leq x < NNZ\}$	$\forall jj1, jj2 : jj1 < jj2 \iff colptr(jj1) \leq colptr(jj2)$ $\forall k1, k2 : k1 < k2 \iff ii * NC + row(k1) < ii * NC + row(k2)$
DIA	$R_{A_{DIA}\rightarrow A_D} = \{[ii, d, jj] \rightarrow [i, j] \mid i = ii \wedge 0 \leq i < NR \wedge 0 \leq d < ND \wedge j = i + off(d) \wedge 0 <= j < NC\}$ $D_{I_{DIA}\rightarrow A_{DIA}} = \{[ii, d, jj] \rightarrow [kd] \mid kd = ND * ii + d\}$	$domain(off) = \{0 \leq x \leq ND\}$	$\forall d1, d2 : d1 < d2 \iff off(d1) < off(d2)$

Table 1. Format Descriptors for COO, CSR, MortonCOO (MCOO), Sorted-COO (SCOO), DIA and CSC.

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. Iterating through COO to retrieve non zeros will have a space $[n, ii, jj]$ and the actual dense coordinate is $[i, j]$. A sparse to dense map of COO is shown in Table 1 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 $\{[n, ii, jj] \rightarrow [n]\}$. The iteration space of CSR is $\{[ii, k, jj]\}$, and its data access relation is $\{[ii, k, jj] \rightarrow [k]\}$. The data access relation decouples the 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 the Table 1

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.

3.2 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 [27]).

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

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.

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

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 [28]. The code synthesis process centers around this mapping.

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

Unknown Uninterpreted Functions. The relation that results from composition in the previous step (in our example $R_{A_{COO}\rightarrow A_{MCOO}}$) 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 has 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 for 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 2.

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.

$$\begin{aligned} R_{A_{src}\rightarrow A_{dest}} &= \{\vec{x} \rightarrow \vec{y} \mid C\} \\ & \vec{x} \in \mathbb{Z}^l,\ \vec{y} \in \mathbb{Z}^r \end{aligned}$$

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 $R_{A_{src}\rightarrow A_{dest}}$ 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

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 2. Across the top of 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.

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 $R_{A_{src} \rightarrow A_{dest}}$ 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 $R_{A_{src} \rightarrow A_{dest}}$ 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 functions. Taken with the universal constraints the relations for P is a function and should be processed first.

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

The resulting relation for P follows.

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

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.

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

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 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. More of this will be this behavior will be discussed in Section 4.

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. Relations that are functions are prioritized. It is possible that the format specification is not precise enough for the relation to be a function. For example, if the destination format is an unsorted COO, the relation will not be a function because there are many UFs that will satisfy the constraints. Relations that are not functions will be chosen only after there are no relations that are functions. By processing them last we ensure that we are satisfying the more-specific constraints and not creating a contradiction.

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 1.

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

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 the 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.

3.3 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.

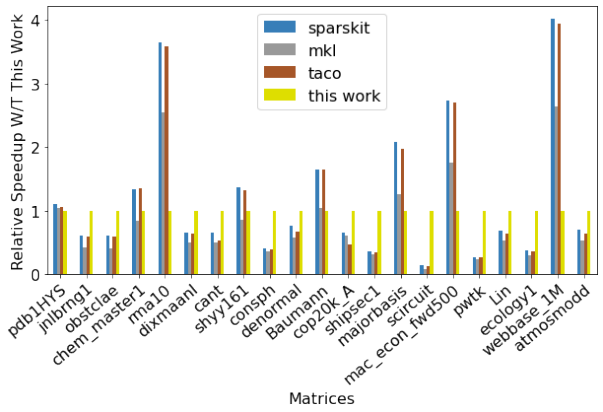
There are situations where the permutation, P is not required for code correctness. This case is detected using standard dead code elimination. SPF, at its most basic, is a dataflow graph. That graph is traversed backward, starting with the live-out dataspaces. Any dataspace and corresponding computation 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 [21] shows support for fusion in SPF. Multiple reads on the same data location occur in synthesis as a series of loop chains.

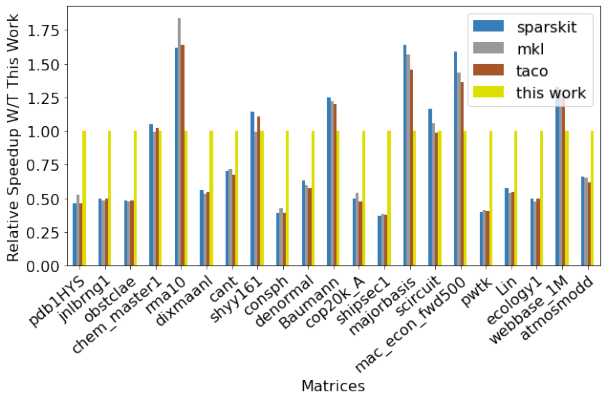
Producer-consumer fusion combines chains of loops that write and then read from the same date. This often results in reducing the space needed for temporary data storage. All opportunities to apply read-reduction and producer-consumer fusion are applied.

4 Evaluation

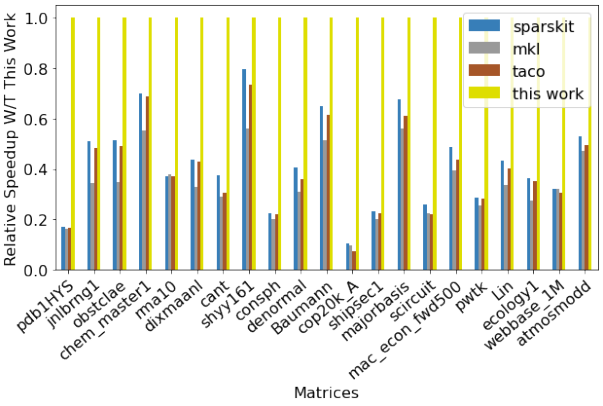
We evaluated the correctness and performance of the synthesized code using a set of sparse matrices from the SuiteSparse Matrix Collection [10]. The synthesized code is serial, we do



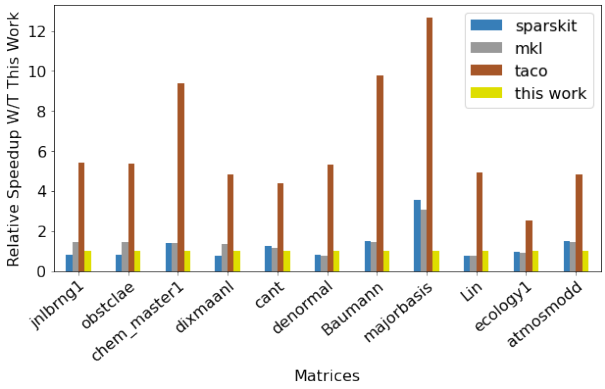
(a) COO to CSC.



(b) CSR to CSC.



(c) COO to CSR.



(d) COO to DIA (W/O Optimization) See Figure 3.

Figure 2. Performance results of generated synthesis code for COO_CSC, CSR_CSC, COO_CSR and COO_DIA. COO is assumed to be sorted lexicographically.

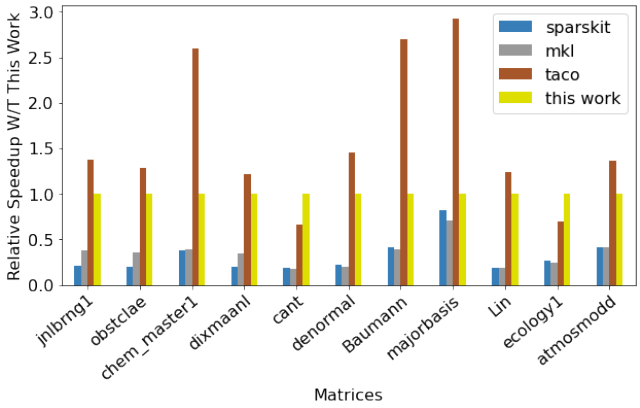


Figure 3. COO to DIA with binary search used to take advantage of monotonicity of synthesized offset array used in copy.

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.

4.1 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.

The performance comparison is made using the same matrix tensors used in TACO's format conversion work. Table 3 shows the matrices used in our evaluation. The COO matrix is assumed to be sorted lexicographically row first. Table 4 shows 3D tensors used in evaluating COO_MCOO reordering.

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 3. Matrices statistics used in evaluating COO_CSR, CSR_CSC,COO_DIA.

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

Table 4. Tensors used in evaluating COO3D_MCOO3.

4.2 Performance Evaluation

We evaluate our algorithm by comparing our results to Intel MKL, TACO [15], SPARSKIT, HiCOO z-morton reordering. Figure 2c 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 2a) and CSR to CSC (Figure 2b) 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.

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. Hand-written z-Morton ordering splits the 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.

Format Description Support			
Tool	Mapping	Re-order	Universal Quantifiers
TACO [15]	✓	×	×
Nandy et. al [20]	×	✓	✓
Venkat et. al [31]	×	✓	✓
This work	✓	✓	✓

Table 5. Automatic sparse format conversion support in our work compared to others.

The permutation data structure enforces reordering constraints in the destination format, however retrieving the permutation (re-ordered position of nonzeros) incurs overhead. This is a limitation in the permutation abstraction implementation and not the approach. 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 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 destination can be done using direct memcopy which we do not explore in this work.

Performance results for COO_DIA in Figure 2d 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 2d) 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 3. Binary search is made possible due to the universal quantifiers on *off* (See Table 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.

5 Related Work

Work most closely related to this includes tensor re-ordering, automatic sparse layout conversion (see Table 5), handwritten sparse layout conversion, and program synthesis.

5.1 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 [18]. Another approach to tensor reordering is to use discrete cosine transform (DCT) to compress Convolutional Neural Networks (CNN) [18]. 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.

5.2 Automatic Sparse Layout Conversion

Table 5 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 facilitates data layout transformations [20, 29, 31]. 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. This means we only need n descriptions. 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.

TACO [8, 9, 15] 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 5.

Bik et al. [4] 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.

5.3 Optimal Tensor Layout

Bik et al. [3] and SIPR [22] optimize computation involving sparse tensors by suggesting more efficient layouts from statically analyzing the computation. Sparso [23] 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.

5.4 Manual Sparse Format Conversion

Sparskit provides various functionalities for dealing with sparse matrices. It helps to translate one matrix form to others [24]. 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 [32]. 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.

5.5 Program Synthesis

Sketching [5, 25, 26] 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 [2] 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. [16] introduced a type system that provides automatic amortized resource analysis to use as a heuristic during the synthesis process.

6 Conclusion

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 framework is unique as it supports tensor formats that rely on re-ordering. A significant limitation of this work is the cost of re-ordering, which introduces a significant overhead. For future work, we intend to introduce more efficient implementations to improve the current performance of our work. Automatically guiding users in selecting the best format is a body of research work that is beyond the scope of this work. This work serves as a foundation for a complete automatic layout transformation for workloads.

References

[1] Khalid Ahmad, Hari Sundar, and Mary Hall. 2019. Data-Driven Mixed Precision Sparse Matrix Vector Multiplication for GPUs. *ACM Trans. Archit. Code Optim.* 16, 4, Article 51 (Dec. 2019), 24 pages. https://doi.org/10.1145/3371275

[2] 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. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

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

[4] Aart J.C. Bik. 2021. Compiler Support for Sparse Tensor Computations in MLIR. https://youtu.be/x-nHc3hBxHM https://youtu.be/x-nHc3hBxHM.

[5] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 775âĂŞ788. https://doi.org/10.1145/2837614.2837666

[6] Chun Chen. 2012. Polyhedra scanning revisited. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2012), 499–508. https://doi.org/10.1145/2254064.2254123

[7] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Technical Report 08-897. University of Southern California.

[8] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 123:1–123:30.

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 823âĂŞ838. https://doi.org/10.1145/3385412.3385963

[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[11] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive Linearized Storage of Sparse Tensors. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 404âĂŞ416. https://doi.org/10.1145/3447818.3461703

[12] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive linearized storage of sparse tensors. *Proceedings of the International Conference on Supercomputing* (2021), 404–416. https://doi.org/10.1145/3447818.3461703 arXiv:2102.10245

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

[14] Pugh W Rosser E Shpeisman T Wonnacott D Kelly W, Maslov V. 1995. The Omega Library interface guide. *University of Maryland at College Park Mar 1995* (1995). https://doi.org/10.1145/3168832

[15] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 77:1–77:29.

[16] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253âĂŞ268. https://doi.org/10.1145/3314221.3314602

[17] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, 19:1–19:15.

[18] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and Effective Sparse Tensor Reordering. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. ACM, New York, NY, USA, 227–237.

[19] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Extending Index-Array Properties for Data Dependence Analysis. In *Languages and Compilers for Parallel Computing (LCPC)*.

[20] Payal Nandy, Mary Hall, Eddie C. Davis, Catherine Olschanowsky, Mahdi Soltan Mohammadi, Wei He, and Michelle Mills Strout. 2018. Abstractions for Specifying Sparse Matrix Data Transformations. In *The 8th International Workshop on Polyhedral Compilation Techniques (IMPACT)*.

[21] Tobi Popoola, Ravi Shankar, Anna Rift, Shivani Singh, Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. 2021. An Object-Oriented Interface to The Sparse Polyhedral Library. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 1825–1831. https://doi.org/10.1109/COMPSAC51774.2021.00275

[22] William Pugh and Tatiana Shpeisman. 1998. 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.

[23] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven optimizations of sparse linear algebra. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 247–259. https://doi.org/10.1145/2967938.2967943

[24] Yousef Saad. 1994. SPARSKIT: a basic tool kit for sparse matrix computations.

[25] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems* (Seoul, Korea) *(APLAS '09)*. Springer-Verlag, Berlin, Heidelberg, 4âĂŞ13. https://doi.org/10.1007/978-3-642-10672-9_3

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404âĂŞ415. https://doi.org/10.1145/1168857.1168907

[27] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. 2013. 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 (2013), 61–75. https://doi.org/10.1007/978-3-642-37658-0_5

[28] Michelle Mills Strout, Geri George, and Catherine Olschanowsky. 2012. Set and Relation Manipulation for the Sparse Polyhedral Framework. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

[29] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. *SIGPLAN Not.* 50, 6 (jun 2015), 521âĂŞ532. https://doi.org/10.1145/2813885.2738003

[30] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine Extensions to Polyhedral Code Generation. In *In International Symposium on Code Generation and Optimization (CGO)*.

[31] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*.

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