Abstractions for Specifying Sparse Matrix Data Transformations

Payal Nandy, Mary Hall
University of Utah
50 S. Central Campus Drive
Salt Lake City, UT 84112, USA
payalgn@cs.utah.edu

Eddie C. Davis,
Catherine Olschanowsky
Boise State University
Boise, ID 83725, USA
eddiedavis@u.boisestate.edu

Mahdi Soltan Mohammadi,
Wei He, Michelle Strout
University of Arizona
Tucson, AZ, USA
kingmahdi@email.arizona.edu,
hewei@email.arizona.edu

Abstract

The inspector/executor paradigm permits using runtime information in concert with compiler optimization. An inspector collects information that is only available at runtime; this information is used by an optimized executor that was created at compile time. Inspectors are widely used in optimizing irregular computations, where information about data dependences, loop bounds, data structures, and memory access patterns are collected at runtime and used to guide code transformation, parallelization, and data layout. Most research that uses inspectors relies on instantiating inspector templates, invoking inspector library code, or manually writing inspectors. This paper describes abstractions for generating inspectors for loop and data transformations for sparse matrix computations using the Sparse Polyhedral Framework (SPF). SPF is an extension of the polyhedral framework for transformation and code generation. SPF extends the polyhedral framework to represent runtime information with uninterpreted functions and inspector computations that explicitly realize such functions at runtime. It has previously been used to derive inspectors for data and iteration space reordering. This paper introduces data transformations into SPF, such as conversions between sparse matrix formats, and show how prior work can be supported by SPF. We also discuss possible extensions to support inspector composition and incorporate other optimizations. This work represents a step towards creating composable inspectors in keeping with the composability of affine transformations on the executors.

Keywords sparse computations, inspector/executor, parallelizing compilers

ACM Reference format:
Payal Nandy, Mary Hall, Eddie C. Davis,
Catherine Olschanowsky, and Mahdi Soltan Mohammadi,
Wei He, Michelle Strout. 2018. Abstractions for Specifying Sparse Matrix Data Transformations. In *Proceedings of Eighth International Workshop on Polyhedral Compilation Techniques, Manchester, United Kingdom, January 23, 2018 (IMPACT 2018)*, 10 pages.
DOI: 0000001.0000001

IMPACT 2018 (http://impact.gforge.inria.fr/impact2018)
Eighth International Workshop on Polyhedral Compilation Techniques,
In conjunction with HiPEAC 2018.
January 23, 2018, Manchester, United Kingdom

1 Introduction

Irregular applications such as molecular dynamics simulations, finite element analysis, and big graph analysis rely on efficient computation over sparse matrices or graphs. Such sparse computations reduce data storage and computation requirements by using indirect accesses through index arrays that store only nonzero data elements. For example, consider the canonical sparse matrix vector multiply (SpMV) in Compressed Sparse Row (CSR) matrix format, a variation of matrix-vector multiplication, $Ax = y$, where the matrix A is sparse, and input and output vectors x and y, respectively, are dense. Only the nonzero entries of A are stored, and

```
  for (i=0; i < N; i++)
    for(j=index[i];j<index[i+1];j++)
S0:   y[i] += A[j]*x[col[j]];

```

Listing 1. SpMV code using CSR sparse matrix format.

auxiliary arrays `index` and `row` represent the starting index in A for elements of each row and the corresponding column indices of entries in A, respectively. The number of nonzero elements, NNZ is stored in the last element of `index`.

Historically, compilers have been severely limited in their ability to optimize such sparse computations due to the indirection that arises in indexing and looping over just the nonzero elements. This indirection gives rise to *non-affine* subscript expressions and loop bounds; i.e., array subscripts and loop bounds are no longer linear expressions of loop indexes. In the listing above, `index` and `col` are index arrays used in non-affine loop bounds and array indexes, respectively. In many such codes, the sparse structure of the matrix or graph does not change through all or significant intervals of the computation so can be analyzed infrequently at runtime. In recognition of this property, *inspector-executor* strategies were developed to parallelize, simplify and/or improve the data locality of such computations [2, 9, 14, 15]. At runtime, the inspector code traverses the index arrays to determine data access patterns and their resulting data dependences. This runtime information can then be used to derive transformed schedules [14, 16] and reordered data structures [3, 4, 8, 10, 23] that the executor uses to execute the computation in a more efficient manner.

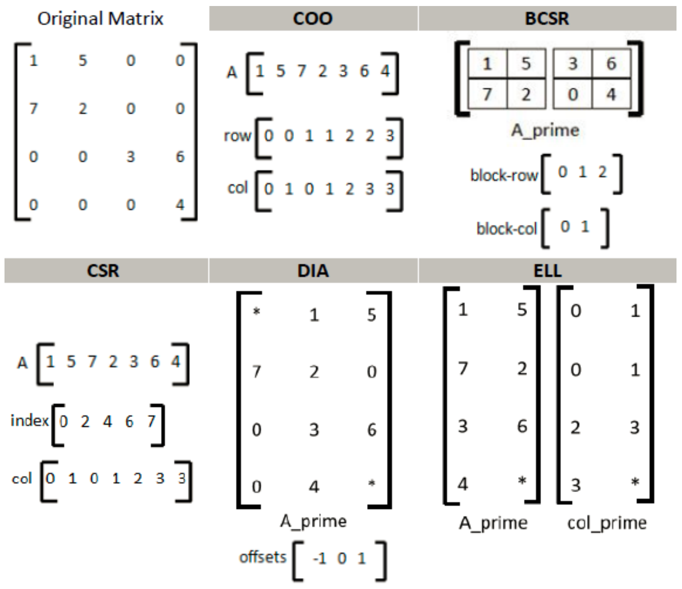


Figure 1. Sparse Matrix Representations

A number of compiler approaches incorporate inspector-executor to perform specific optimizations [9, 12, 17]. Support for non-affine array indexes and loop bounds have been integrated into polyhedral transformation and code generation frameworks through the use of *uninterpreted functions* [6, 13, 18]. Speculative optimizations performed in the Apollo framework [7] apply polyhedral transformations by predicting patterns and verifying at runtime that data dependences have been respected.

Recent work automatically generates the inspector and executor code in a polyhedral framework [20–22]. These all develop custom inspector/executor transformations focused on a single transformation. The optimizations are combined with parallel code generation, and the resulting performance of the generated code has been show to perform very closely, and sometimes faster, than manually-tuned code in libraries such as MKL and OSKI. What is missing is the ability to compose inspector-executor transformations with each other, retaining the composability property that provides the power of polyhedral frameworks in robust code generation in the presence of complex transformation sequences. Formalizing these transformations, which include transformations on sparse data representations, to facilitate inspector composition and generation, is the subject of this paper.

For this purpose, we turn to the Sparse Polyhedral Framework (SPF), which extends the polyhedral framework with uninterpreted functions to represent index arrays at compile time and various function calls that will create those index arrays at runtime [18]. SPF incorporates abstractions for representing runtime data reordering transformations (RTRTs) and iteration reordering transformations for non-affine code. An *Inspector Dependence Graph*, a data flow description that is derived from the loop and data transformations applied to the code, is the abstraction to be used for inspector composition and code generation. The contribution of this paper is to extend SPF to support data transformations that modify the sparse matrix representation to pad with zero-valued elements and expand to include elements that are not present in the original matrix as in [20], and also show how existing support in SPF can represent the iteration-space reordering and RTRTs in [19, 22].

In the remainder of the paper, we describe the automatic derivation of inspectors that unifies this prior work to derive an Inspector Dependence Graph (IDG) and discuss possible extensions to support inspector composition and optimization.

2 Background: Sparse Matrix Formats

As described before, sparse matrix representations only store the nonzero values of the matrix, and have additional auxiliary arrays to record the row and/or column positions. Many different sparse matrix representations have been developed recently to exploit structural properties of the matrices whenever possible to improve code performance. This paper works with the sparse matrix representations shown in Figure1 and described below.

- **Coordinate (COO)** COO maintains a vector `A` of just the nonzeros in the matrix. Auxiliary arrays `row` and `col` are of the same length as `A`, and provide the row and column corresponding to each element of `A` (i.e., its coordinates).
- **Compressed Sparse Row (CSR)** CSR maintains a vector A of just the nonzeros in the matrix. A `col` auxiliary array maintains the column index, the index auxiliary array has one element per row, indicating the index of the first element of that row in the vector `A`.
- **Block CSR (BCSR)** The BCSR format is used when the nonzero values are clustered together in adjacent rows and columns. In the BCSR format, the matrix is divided into small dense blocks containing at least one nonzero element. Zeros are added to make the size of all blocks uniform. The array `A_prime` consists of all such nonzero blocks. The `block-col` auxiliary vector tracks the column of the upper left element of each nonzero block. The `block-.row` auxiliary vector has one element per block row, indicating the index of the first element of that row in the vector `A`.
- **DIA** The DIA format captures only the diagonals that have nonzero elements. The offset auxiliary array represents the offset from the main diagonal. It is well-suited for representing banded matrices.
- **ELL** The ELL format uses a 2-dimensional matrix with a fixed number of nonzero elements per row, and rows with fewer nonzero elements are padded with zero values. An auxiliary col matrix tracks the columns for the nonzero values as in CSR. When most rows have a similar number of nonzero values, ELL leads to more efficient code because

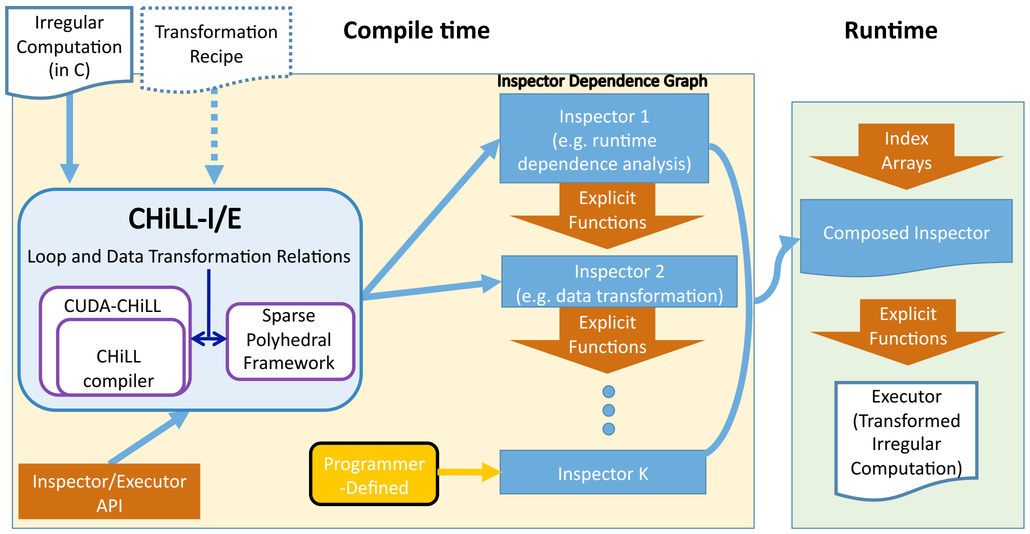

Irregular Computation (in C)
Transformation Recipe
Compile time
Runtime
Inspector Dependence Graph
CHiLL-I/E
Loop and Data Transformation Relations
CUDA-CHiLL
CHiLL compiler
Sparse Polyhedral Framework
Inspector 1 (e.g. runtime dependence analysis)
Explicit Functions
Inspector 2 (e.g. data transformation)
Explicit Functions
Inspector K
Programmer -Defined
Inspector/Executor API
Index Arrays
Composed Inspector
Explicit Functions
Executor (Transformed Irregular Computation)

Figure 2. Overview of CHiLL-I/E framework.

of a fixed number of iterations and no indirection in the loop bounds.

We will use CSR as the default matrix representation, and BCSR, DIA and ELL as exemplars of the range of optimized sparse matrix representations.

3 Overview of Approach

The goal of our approach is automatic generation of both inspector and executor code using extensions to the polyhedral model to represent index arrays with uninterpreted functions. The inspectors then instantiate the uninterpreted functions at runtime, so that they may be consulted by the optimized executor.

These goals are part of a research program called CHiLL-I/E to integrate the existing support for customized inspector generation in CHiLL with more general abstractions for non-affine computations in the Sparse Polyhedral Framework. The project and system under development are depicted in Figure 2. At the heart of the tool are the CHiLL and SPF frameworks, which both reason about loop and data transformations. The input to the tool is a C function and (optionally) a transformation recipe that describes the transformations to be applied. (Our focus has been on designing abstractions and code generation, but some prior work has avoided the need for explicit transformation descriptions using a domain-specific decision algorithm.) The CHiLL compiler and SPF exchange relations that describe loop and data transformations; both are capable of code generation through integration with polyhedra scanning. From the transformations, an inspector dependence graph (IDG) can be automatically constructed, which can then be optimized. The final result is a code that optimizes both inspector and executor. Although both SPF and CHiLL implementations have automated inspector code generation, the extensions to SPF described in this paper have not yet been implemented. Combining forces in CHiLL-I/E will enable more general exploration of inspector code generation and optimization.

For this paper, we concentrate our focus on transformations that modify the data representation, first proposed in [20], extending SPF abstractions for this purpose. This includes conversions from CSR to the other representations listed above, as well as a demonstration of the generality of the approach to codes not operating on sparse matrices with the *Moldyn* example. We use a number of mechanisms that have been presented in prior work but we propose ways to specify these transformations including their inspectors. The specification of these transformations is a step toward the goal of automation.

3.1 Non-affine Iteration and Data Reordering Transformations

In [18, 22, 24], non-affine transformations are defined that require inspectors: coalesce, iteration and data reordering, and iteration space splitting in the presence of non-affine loop bounds, respectively. The transformations all derive run-time relations and perform data transformations in the inspector that are used by the executor. A common feature of the transformations from these prior works is a one-to-one correspondence between iterations in the original iteration space and the transformed iteration space. One-to-one data mappings were also used. An uninterpreted function captures these mappings.

```
for(i = 0; i < N; i++)
  for(k = 0; k < N; k++)
    for(j = index[i]; j < index[i+1]; j++)
      if(k == col[j])
S0:      y[i] += A[j] * x[k]
```

Listing 2. CSR SpMV after make-dense.

3.2 Loop and Data Transformations

Later work presented data transformations composed with standard and non-affine transformations to convert between matrix formations and realize optimized executors [20]. These transformations do not always involve one-to-one mappings. The following non-affine transformations were used for matrix format conversion:

- *make-dense* takes as input a set of non-affine array index expressions and introduces a guard condition and as many dense loops as necessary to replace the non-affine index expressions with affine accesses. The *make-dense* transformation enables further affine loop transformations such as tiling. The intermediate code generated after applying *make-dense* on the code in Listing 1 is shown in Listing 2.
- *compact* and *compact-and-pad* are *inspector-executor* transformations; an automatically generated inspector gathers the iterations of a dense loop that are actually executed and the optimized executor only visits those iterations. The executor represents the transformed code that uses the compacted loop, which can then be further optimized.
- Using *compact-and-pad*, the inspector also performs a data transformation, inserting explicit zeros when necessary to correspond with the optimized executor.

These transformations have been employed for more complex codes not highlighted in this paper, including stochastic gradient descent [5] and Locality-Optimized Block Parallel Conjugate Gradient [1]. We also use similar transformations to support additional matrix representations, such as hybrid CSR+ELL [24] and Compressed Sparse Block [1].

The compact-and-pad transformation adds data locations, for example the zeros to fill out the blocks with at least one non-zero in a block CSR representation. Zeros added into the sparse matrix representation also results in iterations being added to the iteration space. Thus compact-and-pad is not a one-to-one transformation unlike the transformations previously represented in the Sparse Polyhedral Framework.

3.3 Sparse Polyhedral Framework

The Sparse Polyhedral Framework (SPF) builds on the polyhedral framework by representing iteration and data transformations as mappings that can be composed. What the SPF adds is the idea of using uninterpreted functions to represent the index arrays from sparse computations and those produced by the inspector. Thus using integer tuple sets and relations with affine constraints and constraints involving uninterpreted function symbols, it is possible to describe iteration space and data reordering transformations. A composed transformation is a sequence of data and iteration transformation mappings.

Formally, a data reordering transformation is expressed at compile time with a data reordering specification $R_{A\to A\_prime}$, where the data that was originally stored in some location m will be relocated to $R_{A\to A\_prime}(m)$. The compile-time result of reordering an array A is that all access functions with the A data space as their range are modified to target the reordered data space A_prime

An iteration-reordering transformation is expressed with a mapping $T_{I\to I'}$ that assigns each iteration p in iteration space I to iteration $T_{I\to I'}(p)$ in a new iteration space I'. The new execution order is given by the lexicographic order of the iterations in I'.

One of the key ideas in the SPF is that the effect of run-time reordering transformations can be expressed at compile time through formal manipulations of the computation specification (i.e., statement schedules, access functions, and data dependences), thus enabling the compile-time specification of a sequence of RTRTs. The data and iteration reorderings that do not become explicit until runtime are expressed with the help of uninterpreted function symbols. The compile-time generated inspectors will at run-time traverse and construct explicit relations to determine the current state of access functions, scheduling functions, and data dependences and to create reorderings and tilings, which are also stored as explicit relations.

Run-time reordering transformations are typically implemented with inspector-executor strategies. The original code is transformed into the executor code. Although the composed series of transformations can enable an uninterpreted-function-enabled code generator to create the executor code, the inspector must also be specified in some way. In previous work where the inspector-executor transformations specified in SPF were one-to-one mappings, the inspectors were specified with an Inspector Dependence Graphs [19].

The *Inspector Dependence Graph (IDG)* represents various tasks the inspector must perform to generate at runtime explicit versions of what are uninterpreted functions at compile time. The Inspector Dependence Graph (IDG) represents these tasks, the data structures consumed and generated by the inspector, and the dependences between data and tasks within the inspector. The compiler generates the inspector code by walking the Inspector Dependence Graph (IDG).

A set of IDGs are shown in Figures 3 and 4. In the figures, each rectangular node indicates an explicit function, which is the runtime instantiation of an uninterpreted function. Each elliptical node represents a compiler-generated, runtime library, or programmer-provided inspector function.

The next section describes how to specify the transformations and IDGs for data transformations that are not one-to-one mappings.

4 Case Studies

This section describes how the inspectors from [18, 20, 22] can be represented with Inspector Dependence Graphs. In [20], Venkat, Hall, and Strout presented the *make-dense*, *compact*, and *compact-and-pad* transformations as an approach to automatically transform between sparse matrix formats. These transformations were specified as CHiLL script commands and the before and after loop patterns were presented in the paper. The relations arising from the affine and non-affine polyhedral transformations applied to the code provided the foundation for the inspectors that were generated for this work. However, the optimized inspectors were customized for the specific transformations that were applied, and themselves were not composable. Similarly, other inspector-executor approaches based on the polyhedral model show generation of inspectors for specific transformations [18, 22, 24].

In this section, we show how to combine these non-affine loop and data transformations with an Inspector Dependence Graph such that the inspectors can be generated using higher-level compiler abstractions that facilitate composition and optimization of inspector and executor computations. We show for the first time the entire sequence of transformation relations, and in some cases as we generalize, we adjust the code generation strategy. This section derives Inspector Dependence Graphs for a large collection of examples from prior work. We summarize the transformation relations and inspector functions in Table 1, and the resulting Inspector Dependence Graphs in Figures 3 and 4.

4.1 Moldyn

First we show part of an example from previously published work, where the data transformation mappings are one-to-one [19]. This example describes data and iteration reordering for improving data locality. Indirect memory access pattern can negatively impact both spatial and temporal data locality. This is because, they would allow accessing non-contiguous portion of memory, and would make the relationship between iteration ordering and data accesses ambiguous. To reduce such a negative impact, we could apply a data reordering transformation followed by a iteration reordering, a combination suggested in optimizing compiler literature [3, 4]. To demonstrate this, we use a simplified code fragment derived from the molecular dynamics benchmark Moldyn [11] which appeared as Figure 1 in [18] and is repeated for clarity in Listing 3. Table 1 shows that we apply a data reordering and an iteration reordering transformation, the reason of which is to improve data locality of x and fx by removing the indirect accesses in the e loop in the Simplified Moldyn code. Here, we can use a data reordering transformation to improve spatial data locality by bringing the dependent pieces of x and fx closer in memory. The transformation can be done based on a heuristic found in the literature. For instance, we use consecutive packing (cpack) [3] that is generating the reordering function σ for us.

Unfortunately, such data reordering would introduce indirect memory accesses in the i and k loops for x and fx arrays. We can get rid of these newly introduced indirections by reordering iterations of the i and k loops using the same reordering function, σ, used to reorder data in x and fx arrays. Also, note these two transformations introduce other indirect accesses to the vx array in the i and k loops, but for the sake of simplicity we refrain from explaining further transformations that can be applied to remove those indirect accesses or further optimize the code. The Inspector Dependence Graph in Figure 4 shows the process described above.

4.2 CSR to COO

The first row of Table 1 and Figure 3a describe the generation of the COO representation described in [22]. The starting point for the IDG I is the iteration space for the original SpMV code, *index* is an uninterpreted function representing the corresponding array in the CSR format, where `i` is the index into the `index` array and `j` into the `A` and `col` arrays. The coalesce transformation maps the tuple (i, j) to the single iterator k that iterates over the NNZ nonzeros, where $NNZ = count(I)$. Through $c = order(I)$, the inspector creates a runtime mapping `c(i,j)` that maps each integer tuple to a one dimensional lexicographical ordering. The `invert` function reverses that mapping, converting the scalar values of c back to an integer tuple where $c\_inv(k, 0) \rightarrow i$, and $c\_inv(k, 1) \rightarrow j$. These functions are converted from the Inspector Dependence Graph into inspector code at compile time. I_{exec} describes the iteration space of the executor, which for this example is the one-dimensional `k` loop.

4.3 CSR to DIA

We now show how to incorporate data transformations into SPF to derive the code that converts from CSR to DIA format. DIA is appropriate for diagonally dominant matrices. As shown in Figure 1, each diagonal that contains at least one non-zero value of the original matrix `A` is represented as a column in the DIA format. An auxiliary array indicating the diagonal to which the column corresponds accompanies the DIA matrix. We chose DIA for a detailed study because we must pad the matrix in two ways: (1) we introduce zero-valued elements into the matrix for diagonals with some nonzero entries; and, (2) we introduce some additional entries into the matrix (denoted by a $\star$) that do not correspond to matrix elements for diagonals that are shorter than the main diagonal. In practice, both are initialized to 0, but from an iteration space perspective, they are different. The transformation from CSR to DIA therefore creates an `A_prime` that is two-dimensional, with each column representing a diagonal. It must also create an auxiliary matrix called `offsets` that identifies which diagonals each column represents as

Table 1. Details of Inspector Generation

Code in Listing 1 Given: $N \times N$ matrix, $\forall j, 0 \leq col(j) < N$ Original iteration space: $I = \{[i,j] \mid 0 \leq i < N \wedge index(i) \leq j < index(i+1)\}$		
Matrix Format	**Loop and Transformation Specifications**	**Transformation List**

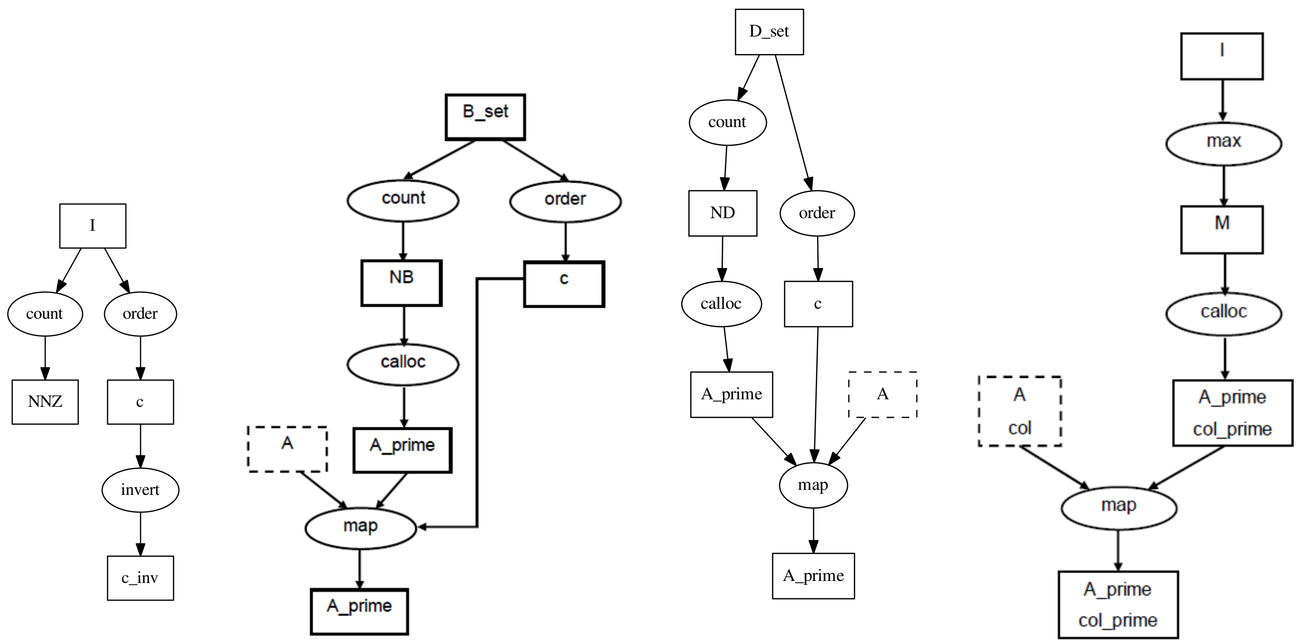
COO	$T_{coalesce} = \{[i,j] \rightarrow [k] \mid k = c(i,j) \quad 0 \leq k < NNZ\}$
	$I_{exec} = T_{coalesce}(I)$
	/* Generate Inspector */
	$NNZ = count(I)$
	$c = order(I)$
	$c\_inv = invert(c)$
		coalesce(S0,k,NNZ,{i, j},c)
		/* coalesced_index = k
		indexes = {i, j}
		inspector function = c */

DIA	$T_{make-dense} = \{[i,j] \rightarrow [i,k,j] \mid 0 \leq k < N \wedge k = col(j)\}$
	$T_{skew} = \{[i,k,j] \rightarrow [i,k',j] \mid k' = k - i\}$
	$T_{compact-and-pad} = \{[k',i,j] \rightarrow [i,d] \mid 0 \leq d < ND \wedge k' = col(j) - i \wedge c(d) = k'\}$
	$I_{exec} = T_{compact-and-pad}(T_{skew}(T_{make-dense}(I)))$
	/* Generate Inspector */
	$D_{set} = \{[k'] \mid \exists j, k' = col(j) - i \wedge index(i) \leq j < index(i+1)\}\}$
	$ND = count(D\_set)$ and $c = order(D\_set)$
	$A_{prime} = calloc(N * ND * sizeof(datatype))$
	$R_{A \rightarrow A\_prime} = \{[j] \rightarrow [i,d] \mid 0 <= d < ND \wedge \exists k', k' = col(j) - i \wedge c(d) = k'\}$
		make_dense(S0, j, k)
		skew(S0, k,[-1,1])
		compact-and-pad(
		S0, k, A, A_prime)

BCSR	$T_{make-dense} = \{[i,j] \rightarrow [i,k,j] \mid 0 \leq k < N \wedge k = col(j)\}$
	$T_{tile} = \{[i,k,j] \rightarrow [ii,kk,ri,ck,j] \mid R * ii + ri = i \wedge 0 \leq ri < R$
	$\wedge\ 0 \leq ii < N/R \wedge C * kk + ck = k \wedge 0 \leq ck < C \wedge 0 \leq kk < N/C\}$
	$T_{compact-and-pad} = \{[ii,kk,ri,ck,j] \rightarrow [b,ri,ck] \mid 0 \leq b < NB$
	$\wedge\ b = c(ii,kk) \wedge\ \exists kk, j \mid kk = \lfloor col(j)/C \rfloor \wedge ck = col(j) - kk * C\}$
	$I_{exec} = T_{compact-and-pad}(T_{tile}(T_{make-dense}(I)))$
	/* Generate Inspector */
	$B\_set = \{[ii,kk] \mid \exists j, kk = \lfloor col(j)/C \rfloor$
	$\wedge\ index(ii * R + ri) \leq j < index(ii * R + ri + 1)\}$
	$NB = count(B\_set)$ and $c = order(B\_set)$
	$A\_prime = calloc(R * C * NB * sizeof(datatype))$
	$R_{A \rightarrow A\_prime} = \{[j] \rightarrow [b,ri,ck] \mid b = c(ii,kk) \wedge\ ck = col(j) - \lfloor col(j)/C \rfloor * C$
	$\wedge\ c(b) = ii, kk\}$
		make_dense(S0,j,k)
		tile(S0, i, R)
		tile(S0, k, C)
		compact-and-pad(
		S0,k,A, A_prime)

ELL	$T_{shift} = \{[i,j] \rightarrow [i, j - index(i)]\}$
	$T_{tile} = \{[i,j] \rightarrow [i,jj,j'] \mid 0 \leq j' < M \wedge M * jj + j < index(i+1) - index(i)\}$
	$T_{compact} = \{[jj,i,j'] \rightarrow [i,j']\}$
	$I_{exec} = T_{compact}(T_{tile}(T_{shift}(I)))$
	/* Generate Inspector */
	$A\_prime = calloc(M * N * sizeof(datatype))$
	$col\_prime = calloc(M * N * sizeof(indextype))$
	$R_{A \rightarrow A\_prime} = \{[j] \rightarrow [i,j'] \mid index(i) \leq j < index(i+1) \wedge j' = j - index(i)\}$
	$R_{col \rightarrow col\_prime} = \{[j] \rightarrow [i,j'] \mid index(i) \leq j < index(i+1) \wedge j' = j - index(i)\}$
		shift(S0, j, j-index(i))
		tile(S0,j,M)
		compact-and-pad(
		S0,M,{A_prime,Col_prime})

Data and Iteration reordering (Moldyn)	$A_{I_e \rightarrow x0} = \{[s,e] \rightarrow [d] : d = left(e)\} \cup \{[s,e] \rightarrow [d] : d = right(e)\}$ $R_{x_0 \rightarrow x_1} = \{[d] \rightarrow [d'] \mid d' = \sigma(d)\}$ $R_{fx_0 \rightarrow fx_1} = \{[d] \rightarrow [d'] \mid d' = \sigma(d)\}$ $T = \{[s,i] \rightarrow [s,i1] \mid i1 = \sigma(i)\}$ $T = \{[s,k] \rightarrow [s,k1] \mid k1 = \sigma(k)\}$ /* Generate Inspector */ σ = cpack($A_{I \rightarrow x0}$) x1 = reorder(x0, σ) fx1 = reorder(fx0, σ) $\sigma^{-1} = inverse(\sigma)$	d_reord(x0, $A_{I \rightarrow x0}$, x1, $A_{I \rightarrow x1}$, σ) d_reord(fx0, $A_{I \rightarrow fx0}$, fx1, $A_{I \rightarrow fx1}$, σ) i_reord(I_i, 1, σ) i_reord(I_k, 1, σ)



(a) IDG for COO format. **(b)** IDG for BCSR format. **(c)** IDG for DIA format. **(d)** IDG for ELL format.

Figure 3. Inspector Dependence Graphs for conversion of a matrix from CSR to different formats.

```
for ( s=0; s < nS ; s++) {
  for ( i=0; i < nV ; i++) {
    x[i] += .. fx[i] .. vx[i] ..;
  }
  for ( e=0; e < nE ; e++) {
    fx[left[e]] += .. x [left[e]] .. x[right[e]]
    ↪ ..;
    fx[right[e]] += .. x[left[e]] .. x[right[e]]
    ↪ ..;
  }
  for (k=0; k < nV ; k++) {
    vx[k] += .. fx[k] ..;
  }
}
```

Listing 3. Simplified code from Moldyn

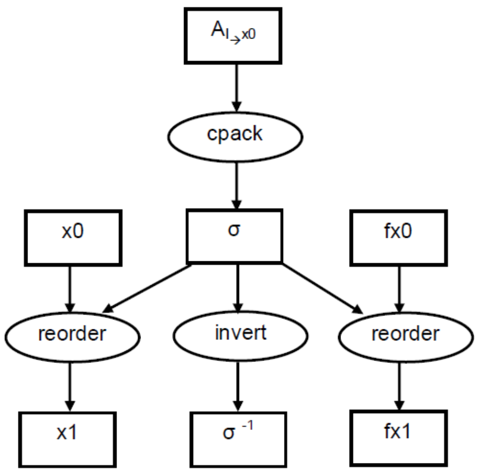
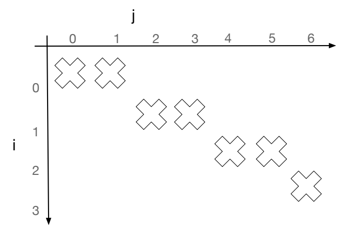


Figure 4. IDG for Moldyn

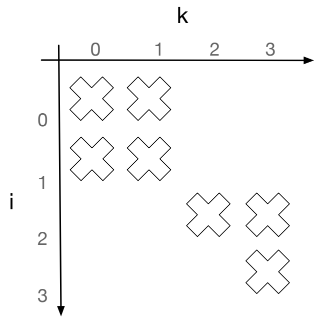
an offset from the main diagonal. These new data representations are created by an inspector and then accessed in the optimized executor.

We derive the interrelated transformations to the iteration space and data representations for the CSR to DIA conversion. The initial iteration space *I* for the CSR code in Listing 1 is illustrated in Figure 5a. Looking at the transformations in the third column of Table 1, we apply *make-dense* and *skew*. The result of *make-dense* is shown in Listing 2 and illustrated by Figure 5b. On this code, we perform an affine skew to move the diagonals to be lined up vertically (all `A[j]` entries such that `col(j)+i = k_prime` for some value of `k_prime`). We arrive at the intermediate code in Listing 4, and illustrated in Figure 5c. The new iterator `k_prime` represents the diagonals in a dense version of the matrix, while retaining the conditional statement to check for nonzero entries in each diagonal.

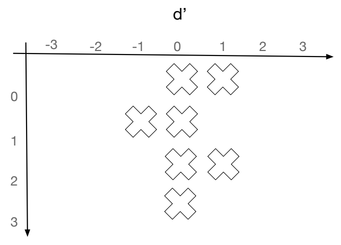
Now, we must apply the `compact-and-pad` transformation that in prior work achieves a number of goals [20]. Here we attempt to use existing and proposed abstractions in SPF



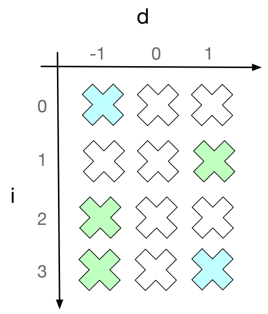
(a) Original iteration space I



(b) I'=T_make-dense(I)



(c) I”=T_skew(I')



(d) I_{exec}=T_cp(I”)

```
  for (i=0; i < N; i++)
    for (k_prime=-i; k_prime < N-i; k_prime++)
      for(j=index[i];j<index[i+1];j++)
        if(k_prime==col[j]-i)
S0:     y[i] += A[j]*x[col[j]];
```

Listing 4. $T_{skew}(T_{make-dense}(I))$.

to produce a similar result using the IDG, although the resulting inspectors are not as optimized. In the inspector, it will: (1) compact the k_prime dimension so that only nonzero diagonals are counted (referred to as D_set in Table 1); (2) the matrix size for both A_prime and offsets can then be derived from the count, allocated and initialized to zero; and, (3) then the data transformation is applied, copying appropriate entries from A to A_prime. The executor will use a new iteration space that is based on the count of diagonals derived by the inspector and that matches the dimensions of the transformed data. To summarize, the effect of compact is to create both inspector and executor code. The executor code requires an iteration space transformation that includes

```
 ND = 0; D_set = emptyset;
 for(i = 0; i<N; i++)
  for(j = index[i]; j < index[i+1]; j++) {
     k_prime = col(j)-i;
     if (!marked[k_prime]) {
        ND++;
        D_set = D_set U <k_prime,ND>;
     }
   }
 A_prime = calloc(N*ND*sizeof(datatype));
 c = calloc(ND*sizeof(indextype));
 for(i = 0; i<N; i++)
  for(j = index[i]; j < index[i+1]; j++) {
     k_prime = col(j)-i;
     d = lookup(k_prime,D_set);
     c[d] = k_prime;
     A_prime[i][d] = A[j];
   }
```

Listing 5. Inspectors for DIA.

```
  for (i=0; i < N; i++)
    for(d=0; d<ND; d++)
S0:   y[i] += A[i][d]*x[i+c[d]];

```

Listing 6. Executor for DIA.

additional iterations as compared to the original iteration space to match the additional zero entries in the transformed data (the green entries of Figure 5d pad the zero entries of the diagonals and the blue entries correspond to the ★ entries in Figure 1).

To see how to derive both codes in SPF, let us first look at the two inspectors that derive the new data representation. First, we count the number of nonzero diagonals by compacting the k_prime loop. Recall that k_prime is modified from the original k loop that was added by *make-dense*. Therefore, we can generate an efficient inspector by projecting the k_prime variable from the iteration space, replacing it with its closed form. The boolean array marked tracks unique k_prime values so that each nonzero diagonal is counted only once. This boolean corresponds to the reference to $\exists j$ in the definition of D_set in Table 1.

Note that the executor iteration space is quite different and derived from i and D_set, the set of diagonals that contain at least one nonzero. To see how this is derived, consider iteration space $I'' = [i, k\_prime, j]$ that is the input to $T_{compact-and-pad}$. After creating the explicit set c in the inspector, we have the mapping c that represents the *offset* array of Figure 1. The resulting executor is shown in Listing 6. This derivation is reflected in the Inspector Dependence Graph of Figure 3. Note that this is a slightly different formulation than in [20], which uses a more optimized inspector with a linked list to avoid two passes, and a GPU-specific executor that includes a few additional affine transformations.

```
for(b = 0; b < NB; b++)
  ii = c[b].ii
  kk = c[b].kk
  for(i = 0; i < R; i++)
    for(k = 0; k < C; k++)
     y[ii*R+i] += A_prime[b][i][k] * x[kk*C+k]
```

Listing 7. BCSR Executor to be Generated.

4.4 CSR to BCSR

As described in [20], the inspector for BCSR needs to generate the vector A_prime, which contains all the dense nonzero blocks of size R×C. To do this, the first transformation applied is *make-dense* which introduces a guard condition and dense loops to remove the non-affine index expression and then replaces it with an affine index access. In this case, the dense loop iterates over all the columns and the guard condition checks whether the column contains a nonzero value. To divide the original matrix into dense blocks of size R×C, the rows (i loop) are tiled by R and the columns (k loop) are tiled by C.

To make the inspector have roughly the same performance as the original computation some of the loop iterators are projected to closed form expressions. Specifically, the kk loop is projected and kk is replaced with $\lfloor col[j]/C \rfloor$ to give the index of the column tile. The code generator also projects the ck loop and replaces its references with a function of C and col(j). This leaves the inspector to iterate over the tiled block rows and original j loop.

The inspector should identify nonzero blocks only once. The first time a particular combination of (ii,kk) is identified, it is marked as visited. The set of all blocks identified is B_set. The built-in inspector routines count and order are called on B_set as shown in the IDG in Figure 3b. Count returns the number of blocks NB in B_set and order creates a lexicographic ordering of the block row ii and block col kk index to the monotonically increasing block identifier bid. The inverse of the function c is c_inv which returns the ii and kk indexes for the executor to perform the SpMV calculation as shown in Listing 7. All this is performed by the inspector as result of the *compact-and-pad* transformation. The inspector also allocates the vector A_prime after finding out NB and initializes all elements to 0. A_prime is then populated using the iteration space $[b, ii, ck]$, copying nonzero entries from A into their corresponding location in A_prime.

4.5 CSR to ELL

The goal of transforming to ELL is to derive a fixed row length M that corresponds to the maximum number of non zeros per row. Then, A_prime is conceptually a 2-dimensional array with N rows and M columns. We shift the j loop so that each iteration goes from 0 to $j - index(i)$. We treat M as an uninterpreted function, and tile the j loop by M. The tile controlling loop will then have only a single iteration. M is computed in the inspector as a max of $index(i+1) - index(i)$. Then the inspector copies the non zero elements of each row i to consecutive elements of A_prime, while remaining entries will be 0 as a result of the *calloc*. The IDG for ELL is shown in Figure 3d.

5 Future Work: Optimizing the IDG

The ability to represent inspectors as IDGs as described in this paper and derive these from loop and data transformations is an important step towards the goal of composing and optimizing inspectors, which is the goal of our future work.

In previous work, specific inspector compositions such as the IDG for coalesce in Figure 3a were generated in an optimized way. For the COO IDG, this meant recognizing that the variable counting iterations in the iteration space I and the variable in the order function were equivalent and could be merged. Also, we noticed that only c_inv is needed in the executor, not c, and that invert can be implemented by iterating over the domain of c, which is equivalent to the iteration space I. Therefore, the generated inspector was optimized by merging count, order, and invert into a single loop and only creating c_inv and NNZ as outputs.

Note that the IDG for converting CSR to BCSR or DIA in Figures 3b and 3c share a similar count, order, and invert pattern as the COO inspector. Count and order are depicted as separate operations because we need to know the size before performing the memory allocation. in [20], these inspectors were merged together and were optimized by allocating blocks or diagonals in a linked list to fuse the count and data mapping inspectors and reduce the number of passes over the code. In fact, a comparison with the BCSR inspector and the OSKI sparse matrix library showed that the generated inspector was substantially faster than the OSKI format conversion due to the use of this linked list representation.

We plan to use these observations to drive an approach to optimizing the IDG. To facilitate fusing inspectors, the IDG must also expose the iteration space of each explicit function representing uninterpreted functions, as well as any fusion-preventing dependences between the explicit functions. We also must support custom optimizations such as using a linked list prior to allocating storage to avoid an additional pass over input. Commonly-used explicit functions will be integrated into the framework as needed.

6 Conclusion and Future Work

In this paper, we walked through the set of transformation relations that were used to generate inspectors and executors, and from these derived Inspector Dependence Graphs. Our plan is to extend the Sparse Polyhedral Framework to support the composition of inspectors so that we may reduce the number of passes that are required over the input data. Conceptually, the IDG is an abstraction for representing a variety of inspectors used for optimizing sparse codes. By generalizing how to represent inspectors, we can build a common interface for the compiler to perform such optimizations. We believe this is an important step towards creating a framework where composition of complex transformation sequences is realistic even for sparse executors and their associated inspectors.

References

[1] K. Ahmad, A. Venkat, and M. Hall. 2016. Optimizing LOBPCG: Sparse Matrix Loop and Data Transformations in Action. In *Languages and Compilers for Parallel Computing: 29th International Workshop*. Springer-Verlag, 218–231.

[2] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*.

[3] Chen Ding and Ken Kennedy. 1999. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 229–241.

[4] Hwansoo Han and Chau-Wen Tseng. 2006. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems* 17, 7 (2006), 606–618.

[5] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali. 2016. Synchronization Trade-Offs in GPU Implementations of Graph Algorithms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 514–523. DOI: http://dx.doi.org/10.1109/IPDPS.2016.106

[6] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega Calculator and Library, version 1.1.0. (November 1996).

[7] Juan Manuel Martinez Caamao, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. 2017. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4192–n/a. DOI: http://dx.doi.org/10.1002/cpe.4192 e4192 cpe.4192.

[8] John Mellor-Crummey, David Whalley, and Ken Kennedy. 2001. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *International Journal of Parallel Programming* 29, 3 (2001), 217–247.

[9] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. 1988. Principles of runtime support for parallel processors. In *Proceedings of the 2nd International Conference on Supercomputing*. 140–152.

[10] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. 1999. Localizing Non-Affine Array References. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 192–202.

[11] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. 1994. *Supporting irregular distributions in Fortran 90D/HPF compilers*. Final Technical Report UMIACS-TR-94-57.1. University of Maryland at College Park, College Park, MD, USA.

[12] R. Ponnusamy, J. Saltz, and A. Choudhary. 1993. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (Supercomputing '93)*. ACM, New York, NY, USA, 361–370. DOI: http://dx.doi.org/10.1145/169627.169752

[13] B. Pugh and D. Wonnacott. 1994. *Nonlinear array dependence analysis*. Final Technical Report CS-TR-3372. Dept. of Computer Science, Univ. of Maryland.

[14] Lawrence Rauchwerger and David Padua. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI '95)*.

[15] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 65–75. DOI: http://dx.doi.org/10.1145/2688500.2688515

[16] Joel Saltz, Chialin Chang, Guy Edjlali, Yuan-Shin Hwang, Bongki Moon, Ravi Ponnusamy, Shamik Sharma, Alan Sussman, Mustafa Uysal, Gagan Agrawal, Raja Das, and Paul Havlak. 1997. Programming Irregular Applications: Runtime Support, Compilation and Tools. *Advances in Computers* 45 (1997), 105–153.

[17] Joel H. Saltz and Ravi Mirchandaney. 1991. The Preprocessed Doacross Loop. In *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume II: Software*. 174–179.

[18] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An Approach for Code Generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57. DOI: http://dx.doi.org/10.1016/j.parco.2016.02.004

[19] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An Approach for Code Generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57.

[20] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 521–532. DOI: http://dx.doi.org/10.1145/2737924.2738003

[21] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 41.

[22] 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*. ACM, 185.

[23] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13)*.

[24] H. Zhang, A. Venkat, and M. Hall. 2016. Compiler Transformation to Generate Hybrid Sparse Computations. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 34–41. DOI: http://dx.doi.org/10.1109/IA3.2016.011