Polyhedral Modeling of Immutable Sparse Matrices

Gabriel Rodríguez
Universidade da Coruña

Louis-Noël Pouchet
Colorado State University

Abstract

Sparse matrices conveniently store only non-zero values of the matrix. This representation may greatly optimize storage and computation size, but typically prevents polyhedral analysis of the application due to the presence of indirection arrays. While specific strategies have been proposed to address this limitation, such as the Sparse Polyhedral Framework and Inspector/Executor approaches, we aim instead in this paper to partition the irregular computation into a union of regular (polyhedral) pieces which can then be optimized by off-the-shelf polyhedral compilers.

This paper proposes to automatically analyze sparse matrix codes with irregular accesses and build a sequence of static control parts which reproduce the sparse matrix computation but using only affine loops and indices. Specifically, we focus on immutable sparse matrices, and develop a custom polyhedral trace compression technique to uncover regular pieces on the input sparse matrix. We illustrate how to build a "sparse" matrix-vector multiplication operation as a union of dense polyhedral subcomputations, and show experimental results for the Harwell-Boeing collection.

ACM Reference Format:
Gabriel Rodríguez and Louis-Noël Pouchet. 2018. Polyhedral Modeling of Immutable Sparse Matrices. In *Proceedings of Eighth International Workshop on Polyhedral Compilation Techniques (IMPACT 2018).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Sparse matrices lie at the core of many different types of applications ranging from data analysis to physics simulation. Their advantage is to allow a 2D indexing of each matrix element (as in a typical dense $N \times M$ matrix), but without actually using storage for elements which equals 0. This can greatly reduce the memory space required to store the useful data in memory compared to a $N \times M$ dense array, and improve performance by skipping operations on the zero elements of the matrix which would not contribute to the final result.

However the optimization of these codes is complex, as compressed representations of sparse matrices such as Compressed Sparse Rows (CSR) typically use arrays to store the coordinates of each non-zero element and another array to store the actual data, leading to indirect array accesses (e.g., `A[B[i]]`). As a result, computations operating on such sparse representation are non-polyhedral programs.

Strategies have been proposed to address this limitation, such as the Sparse Polyhedral Framework [12, 22, 23] and Inspector/Executor (I/E) approaches, e.g., [1, 4, 8, 16, 18] that allow to analyze and optimize important sparse computations using a modified polyhedral compilation framework [22]. However, these approaches do not attempt to exploit the possible regularity in the sparse computation. In the most extreme case, a sparse matrix may be made of only non-zero elements, and thus be totally equivalent to a dense matrix and the sparse computation could be equivalently replaced by a dense-matrix computation, which for many linear algebra operations would be a polyhedral program. That is, in such case, we could apply off-the-shelf polyhedral program optimization techniques, without resorting to sparse abstractions or I/E.

In this paper, we study the occurence of regularity in sparse matrices, with the objective of capturing set(s) of non-zero coordinates as polyhedra. This enables off-the-shelf polyhedral computation on sparse matrices without resorting to any specific framework. We achieve this by employing a two-step approach where first the sparse matrix is analyzed and re-compressed in a polyhedral format at compile-time, followed by emitting a polyhedral program for the computation that is valid only if the sparsity structure does not change (i.e., which elements are non-zeros in the matrix does not change, the sparsity pattern is immutable).

Our approach builds on a powerful mechanism for affine reconstruction of traces [17], which takes a set of integer points as input and computes $\mathcal{Z}$-polyhedra that contain these points. By applying this reasoning on the non-zero coordinates of a given sparse matrix, we rebuild a polyhedral representation of it. A key challenge is to control the reconstruction algorithm, trading off dimensionality for number of pieces: what appears to be disjoint pieces in a 2D layout may be a single piece in a 3D (or more) space, as by raising dimensionality we can capture more complex patterns in a convex way, as shown in this paper.

We make the following contributions:

- We present a Sparse-to-Polyhedral compression technique which converts a *given* sparse matrix (in any format) into a union of polyhedra.
- We show how to use this polyhedral representation to generate affine-only implementation of (sparse immutable) matrix-vector product.
- We evaluate our approach on the 292 sparse matrices from the Harwell-Boeing suite in SuiteSparse Matrix Collection [9] and provide reconstruction statistics.

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

- We apply and evaluate our proposed flow on the SpMV computation, and study criteria for achieving improved performance with our approach.

The paper is organized as follows. Sec. 2 provides background and overview of the approach. Sec. 3 details the mechanisms for the automatic extraction of regular pieces from the irregular sparse matrix. Sec. 4 describes the synthesis of affine programs operating on the compressed matrix. Sec. 5 evaluates our approach on SpMV codes. Related work is discussed in Sec. 6 before concluding.

2 Background and Overview

Dense vs. sparse representation We illustrate our approach by focusing on computing $\overrightarrow{y} = A.\overrightarrow{x}$, a matrix-vector product where the matrix A is read-only, that is A is immutable. A is of size $N \times M$, the coordinate (i, j) identifies the position of an element in A with $0 \leq i < N$, $0 \leq j < M$. When A is a dense matrix, a computation will typically iterate over all values (i, j). When A is stored as a sparse matrix, only non-zero elements are stored and accessible, that is only a subset of coordinates (i, j) in the $N \times M$ grid are represented. When there is a high amount of zero elements in a matrix (an extreme case being a diagonal matrix), significant storage space can be saved by modeling only non-zeros.

```
1|   for (i = 0; i < n; ++i) {
2|I:  y[i] = 0;
3|     for (j = pos[i]; j < pos[i+1]; ++j)
4|S:    y[i] += A_data[j] * x[cols[j]];
5|   }
```

(a)

```
1|   for (i = 0; i < n; ++i) {
2|I:  y[i] = 0;
3|     for (j = 0; j < m; ++j)
4|       if (A_dense[i][j] != 0)
5|S:      y[i] += A_dense[i][j] * x[j];
6|   }
```

(b)

Figure 1. Matrix-vector product

Fig. 1 shows two implementations of this operation: Fig. 1a shows a typical CSR implementation, using `A_data` to store the non-zero data, and `pos` and `cols` arrays to compute/retrieve the coordinates (i, j) associated to a non-zero element and its data value. Fig. 1b shows a similar implementation but using a classical dense representation, where we skip "useless" operations (multiplication by 0), it performs the same number of scalar operations as the CSR code. Using over-approximations [3] and/or uninterpreted functions [20] allow to build a polyhedral representation of these codes, however these approximations do not capture the exact set of non-zero elements of a given matrix and therefore the exact set of operations, as they only exploit static analysis information. Our objective in this work is to bridge this gap, by using compile-time "inspection" of the input sparse matrix, to *generate polyhedral code that is specific to the non-zeros of a given input matrix*, in turn allowing exact polyhedral representation and optimization for this specific sparse matrix. An immediate consequence of our approach is that the code we generate is not valid for any sparse matrix but only for the (immutable) input one, in contrast to SPF and I/E techniques which are general to any input sparse matrix.

From sparse matrix to polyhedra Our approach works as follows. We take as input a matrix (there is no restriction on the input format) and scan it to output a trace of all the (i, j) coordinates which are non-zero. We then attempt to rebuild as few polyhedra as possible to capture all points in this trace. In that sense, our problem has analogy with affine trace reconstruction [17] where the purpose is to rebuild a polyhedral representation of a sequence of addresses being accessed. Let us illustrate with a simple diagonal matrix where only elements (i, i) (on the diagonal) are non-zero, where $N = M = 1000$. The polyhedron describing the non-zero elements is $\mathcal{D} : \{[i, j] : 0 <= i < 1000 \wedge i = j\}$. Once $\mathcal{D}$ is built, the set of (i, j) values to operate on for this matrix is known. Provided an affine implementation of the equivalent dense operation (e.g., Fig. 1b minus the `if` in line 4) is available, by constraining the program iteration domain using $\mathcal{D}$ we can create a conditional-free SCoP for this matrix, and optimize it with off-the-shelf tools. Fig. 2 shows the code specialized to this diagonal matrix, a purely polyhedral program.[1]

```
1|   for (i = 0; i < 1000; ++i) {
2|I:  y[i] = 0;
3|     for (j = i; j <= i; ++j)
4|S:      y[i] += A_poly[i] * x[j];
5|   }
```

Figure 2. Specialized matrix-vector product

Numerous difficulties arise with this approach. First, we must ensure it is always possible to recover the (i, j) indexing to enable integration inside a polyhedral program, yet we do not want to constrain the reconstructed structure to be 2D: Fig. 2 uses a 1D array for example, which is sufficient to capture in a single domain all non-zero elements here. Second, we must control the number and complexity of the polyhedra being re-built to describe the sparse matrix. An extreme case where each non-zero is captured in a single polyhedron (one point in it) is always possible, yet would be practically useless. There is a trade-off between the complexity of such polyhedra (and therefore the complexity of

[1] The `j` loop is shown for illustration purpose, and is not present in the code we actually generate as it is eliminated by CLooG since it only iterates once.

the code scanning them to be generated by CLooG [2] here) versus the overhead of indirect array accesses in the original sparse representation.

Specifically, we build on the affine Trace Reconstruction Engine (TRE) [17] and modify it to consider the above trade-off by controlling the dimensionality and size of polyhedra reconstructed, while preserving the ability to perform (i, j) indexing. This tool employs an algebraic approach which, taking as input the trace of memory addresses accessed by a single memory reference, synthesizes an affine loop with a single perfectly nested reference that generates the original trace. The tool has been extended to: i) support the synthesis of unions of affine loops; ii) analyze several address streams in parallel, synthesizing multiple accesses inside a single loop; and iii) support the reconstruction of complex streams as a sequence of affine statements. The synthesis of affine codes from the traces of sparse computations is described in depth in Sec. 3.

Proposed workflow The end-to-end workflow we propose is as follows.

1. The user isolates a sparse matrix based computation of interest, such that (a) the sparse matrix are immutable; and (b) a dense-matrix version of the computation is known, and is a SCoP.
2. A trace of all non-zero coordinates is produced, by scanning the sparse matrix.
3. Affine trace reconstruction is run on the trace to build sets of polyhedra that model the non-zero coordinates.
4. The dense-matrix version of the code (i.e., which operates on all $N \times M$ matrix elements) is restricted to only execute operations associated with a non-zero entry in the sparse matrix. In the case of SpMV as studied in this paper, as there is one "operation" per non-zero element, this simply amounts to scanning the rebuilt set of polyhedra.
5. The polyhedral representation of this matrix-specific program can then be optionally optimized by polyhedral compilers, or directly code-generated with CLooG.

We insist that our flow is *specific to a particular sparse matrix*, that is the code we generate is valid only for this matrix, which in addition must be sparse-immutable (its non-zero structure does not change). In that sense it is best suited for scenarios where the same large matrix is accessed many times (e.g., from within an iterative loop, as with PageRank).

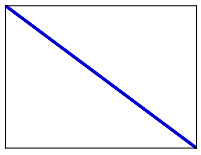
3 Polyhedral Sparse Matrix Compression

In the following, we present our approach to creating a polyhedron-based description of a sparse matrix. To better illustrate the process along with how the computation is performed, we focus on the SpMV operation, as shown in Fig. 1a. Indeed, for matrix-vector product (assuming the output vector $\overrightarrow{y}$ is set to 0 prior to it) there is a single statement performing the computation, and there is as many executions of the statement as there are non-zero in the sparse matrix. That is, the process of building polyhedra describing the sparse matrix is equivalent to building polyhedra describing subsets of the iteration domain that need to be executed. Sec. 4 discusses the more general case.

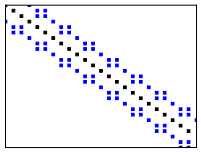
Sparse compression as trace reconstruction Consider the sequence of accesses in Fig. 3a, corresponding to executing the SpMV code in Fig. 1a using the matrix `HB/nos1` from the SuiteSparse Matrix Collection [9][2]. A convenient feature of SpMV computation is that the (i, j) coordinates of each non-zero is explicitly built to access the vectors, in other words tracing the values of `i` and `cols[j]` gives exactly the (i, j) coordinates at which non-zeros exist.

i	cols[j]	&(A_data[j])
0	0	0x00
0	3	0x04
1	1	0x08
1	4	0x0C
1	5	0x10
2	2	0x14
2	4	0x18
2	5	0x1C
3	0	0x20
3	3	0x24
3	6	0x28
	⋮	

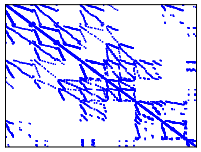
(a)



(b)



(c)



(d)

Figure 3. Different sparse matrices from the HB group of the SuiteSparse Matrix Collection. Fig. a) shows an excerpt of the accesses performed during SpMV of matrix `HB/nos-1`. The nonzero elements in this matrix are shown in Fig. b), and a zoom of its main diagonal is provided in Fig. c). This is a 237×237 matrix with $1,017$ nonzero elements, and its SpMV affine equivalent code consists of a single statement inside an 8-dimensional loop. Fig. d) shows the nonzero elements in `HB/can_1072`, a $1,072 \times 1,072$ matrix with $12,444$ nonzero elements which does not exhibit any apparent regularity. Its SpMV affine equivalent code includes 870 statements of up to 8 dimensions.

As can be seen in Fig. 3b, all the nonzero elements lie *nearby* the main diagonal, and zooming on this diagonal in Fig. 3c, we can see upon closer inspection there is a recognizable sparsity pattern. An affine loop traversing all the nonzero positions of this matrix must use piecewise iteration domains, which are not easy to reason about, even by

[2]In the remainder of the paper we will refer to different matrices in the SuiteSparse collection using this `<group>/<matrix>` notation.

domain experts. Instead of following a manual approach, we designed a modified version of the Trace Reconstruction Engine (TRE) [17], an analysis tool that takes as input the sequence of memory addresses issued by an isolated memory reference and generates an affine loop with a single perfectly nested statement that produces the original access sequence. The tool works by iteratively analyzing each address a^k in the trace and building a projection vector $\overrightarrow{c}$ and candidate iteration vectors $\overrightarrow{\imath}^k$ such that: i) $\overrightarrow{\imath}^k$ lexicographically follows $\overrightarrow{\imath}^{k-1}$; and ii) addresses in the trace are matched, i.e., $\overrightarrow{\imath}^k \cdot \overrightarrow{c} = a^k$.

The reader may refer to [17] for details about TRE. In a nutshell, it rebuilds a $\mathcal{Z}$-polyhedron (the intersection of a $\mathbb{Z}$-polyhedron $\mathcal{D}$ and an affine integer lattice F [10]) that captures a stream of addresses: both an iteration domain $\mathcal{D}$ and an affine multidimensional access function F from that domain to the trace element value are being built. But a fundamental aspect of this approach is that it may rebuild an iteration domain $\mathcal{D}$ of arbitrary dimensionality for the purpose of capturing seemingly "distant" points into a dense/-convex polyhedron. It only guarantees that it builds at the same time an access function that, when applied on each point in the rebuilt iteration domain, will produce exactly all addresses in the original trace. In this work, we have a supplementary constraint to handle: we must also for each point in this TRE-reconstructed iteration domain be able to produce the associated i and j value ((i, j) coordinate).

Our approach to handle this case is to extend the TRE tool as follows. The original TRE rebuilds $(\mathcal{D}_A, F_A)$, the reconstructed $\mathcal{Z}$-polyhedron for a single address stream of A. We want instead as output $(\mathcal{D}_A, F_A, F_i, F_j)$ to be reconstructed, where F_i is the function that, when applied to any point in $\mathcal{D}_A$, provides the i coordinate of this point, conversely for F_j to obtain the (i, j) matrix coordinate for any element. We achieve this by modifying TRE to instead analyze three streams simultaneously, with the additional constraint that $\mathcal{D}$ is identical for all three streams, i.e., only the reconstructed F can be different for the three different streams. Precisely, the three streams correspond to building F_A, F_i and F_j each being a stream of scalar values. For each non-zero element, the trace entry has three components: the address `A_data[j]` being accessed, the value of `i`, and the value of `cols[j]`. Note that by reconstructing F_A also, which captures the memory location of the data associated to a particular (i, j) coordinate, *the polyhedral code can operate directly on the input sparse matrix representation (including, but not limited to, CSR).* This means data does not need to be copied in a new location for the program to proceed, avoiding additional storage. Precisely, only `A_data` is needed, the `cols` and `pos` arrays are not used anymore after polyhedral compression.

Rebulding simultaneously F_A, F_i and F_j in a $\mathcal{Z}$-polyhedron ensures we can always rebuild the (i, j) coordinates. One may observe that for SpMV, the evolution of the values of i and j is explicitly captured by the stream of addresses accessed for `y` and `x`. The execution of the original SpMV code is instrumented and the sequence of accesses to arrays `A`, `x` and `y` is traced (see Fig. 3a), and used as input to the TRE.

Extended TRE algorithm The processing starts at the beginning of the trace, trying to model the trace elements using a single iteration domain. This may not be possible in the general case without using an intractably large number of dimensions [17]. For this reason, timeout mechanisms are included to halt the analysis when it does not achieve significant advances after a specified number of steps. In this case, a new statement T_1 is generated using the largest loop found until the timeout. The reconstruction process continues trying to synthesize another statement T_2 starting from trace position $(\#\mathcal{D}^{T_1} + 1)$. The previous process repeats until the complete affine equivalent code is synthesized. Algorithm 1 shows a pseudocode of each step of the TRE. Esentially, the processing starts with an empty SCoP, and tries to enlarge it by sequentially adding points in the trace inside the `add_iter` call in line 14: in line 2 all the indices lexicographically following the most recent one are generated, while the loop in line 3 checks whether each of the generated indices explains the next value $\overrightarrow{\imath}^k$ in the trace. A list of candidate SCoPs is maintained and sorted by fitness heuristics. Whenever a solution cannot be found building over the best ranked candidate, control will return to the `TRE` function, which will retrieve the best ranked candidate in line 11, increase its dimensionality to incorporate one new point to the SCoP, and continue processing it.

The TRE was extended in two different ways. First, in order to reconstruct general piecewise-affine domains we had to support more complex bounds matrices. In its original form, TRE employs a square bounds matrix $\mathbf{U} \in \mathbb{Z}^{D \times D}$, with D the number of dimensions of the loop. Each row j encodes an affine upper bound of the form $i_j \leq u_j(i_1, \ldots, i_{j-1})$. Only upper bounds faces were explicitly encoded, implicitly using $i_j \geq 0$ for all loop indices. In the extended version lower bounds faces are explicit and general, i.e., they encode $i_j \geq l_j(i_1, \ldots, i_{j-1})$. Furthermore, multiple bounds faces can be included at any loop level. As such, the new bounds matrix is not square, but $\mathbf{U} \in \mathbb{Z}^{Fx D}$, with F the number of faces of the iteration polyhedron.

The second extension aims at supporting the synthesis of several statements inside a loop. This is achieved using a rather simple structural extension of the basic algorithm. Instead of focusing on a single stream of addresses, several of them are analyzed in parallel. Line 3 in Algorithm 1 is changed so that an index is valid only if $\overrightarrow{\imath}\mathbf{C} = \overrightarrow{d}^k$, where now $\overrightarrow{d}^k$ is not a single address but a tuple containing the addresses issued in the same iteration for all the streams being analyzed, and each column in matrix $\mathbf{C}$ contains the access functions for each of the analyzed streams.

Complexity trade-offs It is theoretically always possible to rebuild a set of integer points as a union of polyhedra, in the worst case using polyhedra of only one point each. In

Algorithm 1: Pseudocode of the TRE

```
Input: the access trace, 𝒜; an input SCoP S
Output: a SCoP reproducing the accesses in 𝒜
Function add_iter(𝒜, S)
    Find ℒ = {i⃗^k} lexicographically following i⃗^{k-1} ∈ S;
    for i⃗^k ∈ ℒ such that i⃗^k c⃗ = a^k do
        S' = S ∪ i⃗^k;
        S_list = S_list ∪ add_iter(𝒜, S')
    end
end
Function TRE(𝒜)
    S_list = {EMPTY_SCoP};
    while True do
        // retrieve the best ranked SCoP
        S = retrieve_best(S_list);
        if (timed out) or (#D^S == len(𝒜)) then return 𝒮;
        // add dimension to include a^k
        S = increase_dimensionality(S, a^k);
        add_iter(𝒜, S);
    end
end
```

this case the variable which may grow uncontrollably is the number of total pieces s, which is bounded by $\frac{n}{max_d}$, where n is the number of entries per reference in the trace, and max_d is the maximum number of dimensions allowed per polyhedron. The fundamental tradeoff is between number of pieces and dimensionality: a sequence of points which cannot be captured using a 2D affine loop nest may be captured using a 3D loop nest. For instance, consider a 2D-tiled traversal of a 2D $N \times N$ array with block size B. This traversal can be coded as: i) a sequence of $\frac{N}{B} \times \frac{N}{B}$ 2D loops, each of them traversing a $B \times B$ section of the array; or ii) a sequence of $\frac{N}{B}$ 3D loops, each of them traversing a single column (or row) of tiles; or iii) a single 4D loop traversing the full set of tiles. Table 1 shows the number of statements (i.e., pieces) required using different maximum dimensionalities, for rebuilding the `HB/nos1` matrix which exposes a quite complex sparsity structure.

Table 1. Evolution of the number of pieces as a function of their maximal dimensionality (max_d) for matrix `HB/nos1` (1,017 nonzero elements).

max$_d$	2	3	4	5	6	7	8
pieces	312	159	81	4	3	2	1

Table 1 clearly indicates highly different choices are possible, between manipulating a single 8D domain (intuitively this will lead to a 8-deep loop nest to scan the polyhedrally compressed matrix) versus 312 disjoint 2D pieces. Sec. 5 discusses the potential performance impacts of such trade-off.

4 Generating Optimized Programs

We now discuss how the end-to-end program is generated, first specifically for the experiments presented in this paper, and then for more general cases.

Generating affine-SpMV programs As noted in Sec. 3, the matrix-vector product operation has the convenient property of executing exactly one statement instance for each non-zero entry. That is, scanning the set of polyhedra that model the input matrix do correspond exactly to scanning the "actual" iteration domain of the SpMV computation. We perform extensive evaluation in Sec. 5 of such SpMV regeneration, using the following process:

1. Sets of $\mathcal{Z}$-polyhedra modeling the non-zero of the input sparse matrix are reconstructed, as per Sec. 3.
2. A SCoP (in scoplib format) is created by creating a polyhedral statement for each piece, using its polyhedron domain as the corresponding statement iteration domain in the SCoP. For each, a statement text (in C) is defined, using the access functions reconstructed to emit `y[..] += A_poly[..] * x[..]` where `..` is adequately replaced by the expression to compute i, the position of the data in the compressed array, and j.
3. This SCoP is then generated as a C program using the PoCC compiler [15], using CLooG's code generation. As all operations are parallel in this specific case, we do not specify a full schedule and instead let CLooG generate the C code scanning all domains using its own heuristics.

Potential extensions Although we limit in this paper to the study of SpMV computations, our approach can potentially be applied to a variety of computations which have a "dense array" equivalent to the considered sparse computation. Precisely, we require the user to provide the SCoP of the program for the dense version of the kernel of interest, i.e., which computes the same mathematical function and iterates on all $N \times M$ (i, j) elements of the matrix. However a caveat is that for seamless integration by intersecting the program's iteration domain with the union of pieces reconstructed to restrict the program to only operating on the non-zero elements, we must recognize in the SCoP which loop iterators correspond to indexing the (i, j) elements. For typical linear algebra computations these iterators are usually explicit, but this may require more complex matching algorithms for general affine programs. In addition, we mention this is not a "simple intersection" of domains: we need to use the F_i, F_j reconstructed access functions to build the constraints on the SCoP iteration domain, as $\mathcal{D}_A$ may be of arbitrary dimensionality and cannot be intersected as-is with the SCoP's iteration domain.

Using off-the-shelf polyhedral compilers Once a SCoP is built as depicted above and properly adjusted to only execute the relevant operations, it can then be processed by typical polyhedral compilers such as Pluto [5] or PPCG [24]

for example. Typical optimizations for data locality, parallelism and vectorization can be applied.

But an important additional degree of freedom is that in many linear algebra computations there is a vast amount of parallelism available. To an extreme, allowing associative reordering, for matrix-vector product all pointwise operations can be executed in any order. This implies that the order in which polyhedra are scanned does not impact semantics, in turn suggesting that the order in which points in the trace are read will not impact the semantics of, e.g., affine-SpMV codes. By changing the order in which points are processed by TRE we may influence (reduce) the dimensionality of the reconstructed polyhedron. Although we did not exploit this degree of freedom in this paper, it is an important component of our future work.

5 Experimental results

The approach for SpMV described previously has been applied to all the matrices in the Harwell-Boeing matrix repository (the `HB` group of the Suitesparse Matrix Collection). During the reconstruction process, we periodically check whether more than 1,000 pieces are expected to be generated. In this case, the synthesis process is halted and the matrix in question skipped, as we estimate that more than 1,000 pieces (that is, more than 1,000 *statements*[3] in the final SpMV code) is mostly impractical for current polyhedral compilers such as PoCC. Out of the 292 matrices, 235 are reconstructed using at most 1,000 statements. Figure 4 shows the histogram of the number of statements in the generated SCoPs over all 235 reconstructed matrices. Reconstruction time is highly related with the number of generated statements: as we employ a timeout to stop exploring alternate branches for the compression (i.e., trading off number of pieces versus maximal dimensionality), this timeout length drives the reconstruction time in our experiments. The time required per statement was approximately 15 seconds.

Table 2 details some reconstruction statistics about these matrices. Data is aggregated by the number of statements/-pieces required. It shows how the complexity characteristics of the generated loops (the dimension and number of faces of the iteration polyhedron) are essentially invariant when the number of statements increases, which means that the reconstruction heuristics employed by TRE are inherently limiting the maximum complexity of the synthesized loops. The exception are matrices which exhibit a high degree of structural regularity, and can then be reconstructed using a small number of statements. In these cases we observe how the synthesized loops are structurally simpler, and at the same time have much larger iteration domains. Note that the parameters which control maximum complexity could be changed in the TRE, leading to SCoPs with a larger number of simpler statements, or a smaller number of more complex ones.

[3]In this section we use interchangeably the words "piece" and "statement" as the affine-SpMV code is generated by scanning the pieces, leading to one statement per piece.

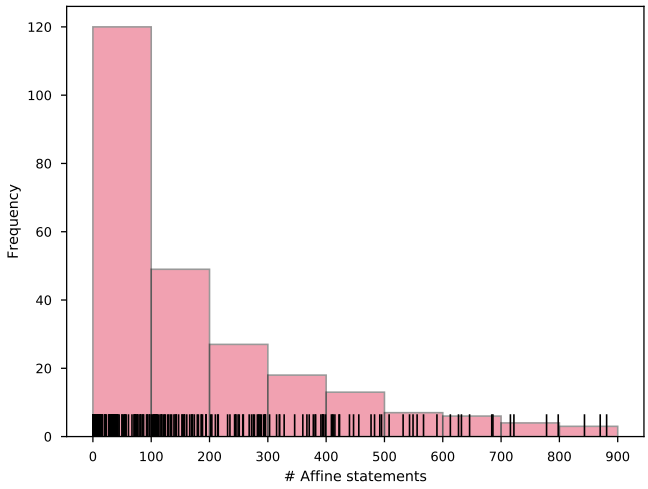


Figure 4. Histogram and rug plot of the number of statements/pieces required to reconstruct matrices in the Harwell-Boeing repository. The minimum number of statements is 1. The maximum number is 881.

Table 2. Statistics for the Harwell-Boeing collection. Data is quantized by the number of statements in each affine-SpMV SCoP (see Fig. 4). The table shows the geometric mean of the number of dimensions of the iteration polyhedron (loop depth), the number of nonzeros of the original matrices (`nnz`), the number of statements necessary for the reconstruction (`stmts`), the number of iterations of the inner loop of each statement (`stmts`), and the number of matrices belonging to each category (`count`).

category	dims	nnz	stmts	iters	count
(0, 5]	2.47	699.56	1.43	489.42	32
(5, 20]	6.39	631.72	11.42	55.29	22
(20, 100]	6.32	1524.51	49.55	30.77	67
(100, 200]	6.29	3560.80	137.73	25.85	48
(200, 400]	6.31	7202.05	293.90	24.51	45
(400, 600]	6.40	8865.98	477.95	18.55	20
(600, 800]	6.16	17984.74	687.62	26.16	10

5.1 Performance of optimized codes

We use PoCC 1.4.1 [15] to compile the generated SCoPs to C code. The compilation chain is mostly limited to code generation (CLooG), but we use AST post-processing in PoCC to simplify and hoist loop bounds, and loop unrolling. Not all the generated SCoPs can be handled by this toolchain: only 179 out of the total 235 are converted to C code. In the remaining 56 cases, CLooG exhausts the 64 GB of available

RAM and is killed. SCoPs with up to approximately 500 statements can be handled in our experiments, which motivates the previous decision to interrupt the synthesis process for matrices requiring more than 1,000 statements.

GNU GCC 7.2.0 is used to compile the generated codes using `-O3`, which includes automatic vectorization. The benchmarks are executed on an Intel Core i7 4790 (Haswell). We run both irregular and affine versions of the 179 different input matrices. Executions are instrumented using hardware counters through PAPI [13]. Each sparse matrix-vector multiplication is performed 1,000 times and the aggregated counter values are reported. We measured performance improvements from 0.10x (i.e., a 10x slowdown) to 6.17x, with an average improvement of 1.38x. After broadly analyzing the impact of different factors, we find the increase in executed instructions count to be the most performance-impacting one. Figure 5 shows the relationship between normalized instruction count and speedup. There is a clear inverse logarithmic relationship between both, and we can broadly divide our matrices into four different categories. We further analyze the reasons for the obtained performance by singling out a few representative benchmarks for each of the three regions with data points (note that no benchmarks achieve better performance than their irregular counterparts when they execute more instructions, so one of the original four regions is empty). A graphical summary of the most relevant performance counters is given in Fig. 6. The performance counter values for the irregular kernels are given in Table 3.

Let us consider first the lower right corner of Fig. 5, containing benchmarks in which instruction count increases and performance decreases. This region includes 40 benchmarks. `HB/nos1` appears in the far right extreme of that region. This benchmark is a perfect example of a class in which loop-related instructions (integer additions, comparisons, the multiplications used in affine access functions, branches) significantly increase the total instruction count of the application. Even memory access increases due to register pressure. This usually happens when the original, irregular 2-level loop is replaced by a more complex dense one. For `HB/nos1`, the affine-SpMV code contains a single statement, but enclosed into an 8-dimensional loop. The L1 cache accesses increase by 9, branch instructions by 7, and other instructions (integer additions, subtractions, multiplications, comparisons, etc. supporting loop control flow) increase by 13. As a result, the code is much slower simply because many more computations need to be carried out, even if cache behavior is excellent (both L1 data and instruction misses are below 0.00001%).

In the lower left corner of Fig. 5 we find benchmarks which execute less instructions than the original irregular code, but still cannot achieve performance improvements, including 19 benchmarks. A paramount example of this class of codes is `HB/jagmesh1`. In this case, the instruction count decreases

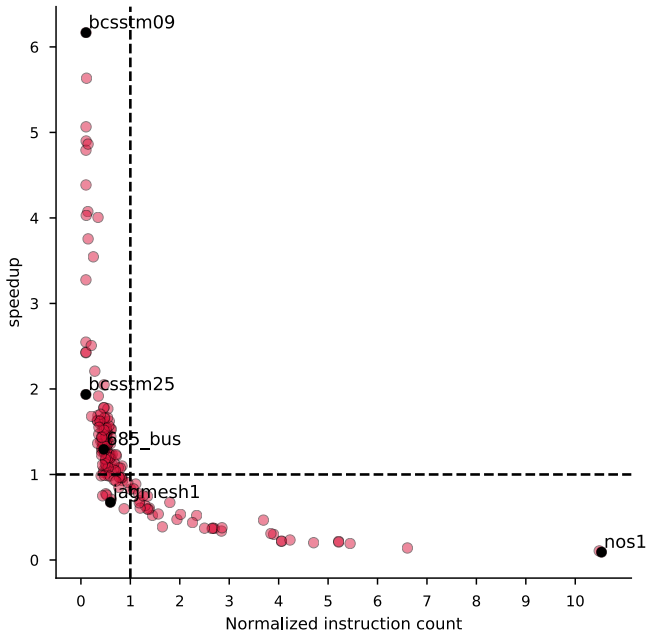


Figure 5. Instruction count of affine-SpMV codes normalized to original SpMV codes vs. speedup. The slashed lines divide the space into four regions. No affine code executing more instructions than its irregular counterpart achieves better performance.

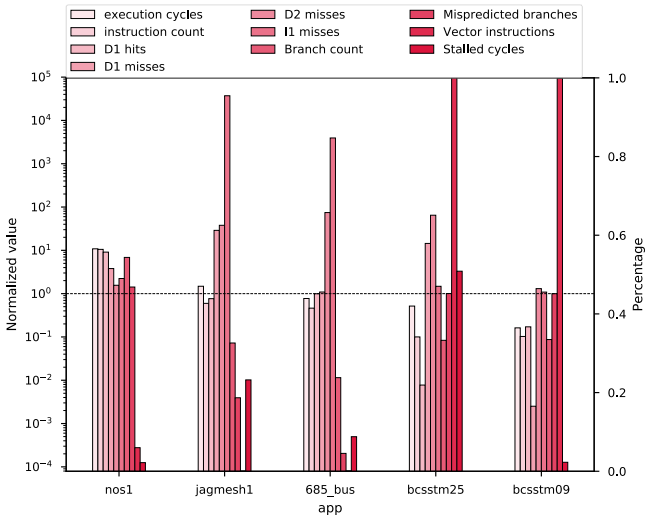


Figure 6. Performance counters for the singled-out benchmarks. Values are normalized with respect to the original irregular code. The number of stalled cycles and of vector operations are given as a percentage of total cycles and floating point operations, respectively, and are referenced to the right axis.

Table 3. Performance counter values for the irregular version of the singled-out benchmarks.

matrix	ex. cyc.	inst. count	D1 hits	D1 misses	D2 misses	I1 misses	Br. count	Mispred. br.	Stalled cyc.
nos1	2776143	9424195	3524280	47	19	32	1492049	1004	104333
jagmesh1	17107678	50692196	20409426	99334	3317	40	8137050	31118	1202684
685_bus	11001499	29088196	8928827	672438	189	40	4620050	48850	1201676
bcsstm25	79524188	308784201	75879071	351066	31069	31	46318055	1004	3056718
bcsstm09	5531743	21664196	4827409	184956	51	31	3250050	1005	181461

significantly, as the affine-SpMV version executes 30M instructions vs. the 51M instructions of the irregular code. Some of the performance counters for this matrix clearly improve: the number of executed branches significantly decreases (0.6M vs 8.1M), as do branch mispredictions (which are anecdotical in both code versions). However, the number of stalled cycles increases by a factor of 5. The culprit seems to be cache behavior: L1 data misses are increased by 29, and instruction misses go from just 40 to 1.5M. The explanation in this case is in the code generated by CLooG: it translates the original 285 statements in the SCoP to a single monolothic outer loop with only two iterations, containing the original accesses in an almost completely unrolled fashion. While this avoids the increase in instruction count observed for `HB/nos1`, it increases the code size by 16.

On the upper left section of Fig. 5 we find benchmarks which execute less instructions than their irregular counterparts, and do so faster. These include the remaining 120 benchmarks. The reasons for success in this region are varied. Extreme cases correspond to completely regular matrices, such as diagonal matrices or matrices with no zeros. These can be traversed using a single 1-dimensional statement. In the extreme case of `HB/bcsstm09` the number of instructions decreases by a factor of 10, and we also observe significant reductions in the number of branches and data accesses. Furthermore, 100% of the floating point instructions are vectorized. The final speedup is 6.17. This matrix contains 1,083 nonzero elements. Other similar diagonal but much larger matrices, such as `HB/bcsstm25` with 15,439 nonzeros, also achieve notable speedups for the same reasons (1.94 in this particular example). The reason for the slower performance in this case is the difference in memory footprints: in the smallest case it is only of 25kB, fitting L1, whereas in the largest one it goes up to 361 kB, which makes temporal locality across the 1,000 repetitions of the SpMV operation be exploited through the L2/L3 caches.

Finally, we draw the attention towards more modest performance improvements. In these cases, the reasons for success are varied, and the result of different tradeoffs in instruction count, memory footprint, and runtime improvements. Consider the case of `HB/685_bus`. The code ends largely unrolled by PoCC/CLooG, using half the number of total executed instructions. However, code size increases, and consequently worsens the behavior of the instruction cache. However, the total number of data memory access is decreased, as is the number of branch mispredictions, for a total improvement of 1.29x.

6 Related Work

Sparse codes characteristically exhibit irregular access patterns to one or more arrays that prevent static code analysis and optimization. Their prevalence in scientific computing, and in particular using distributed-memory clusters lead to the design of inspector/executor (I/E) approaches pioneered by Saltz et al. [18]. They developed runtime infrastructure for distributed memory parallelization of irregular applications [14, 18, 19]. These were augmented with compiler approaches that automatically generated parallel code [1, 8, 25]. Ravishankar et al. exploit run-time regularity to produce polyhedrally-optimizable executor code in specific cases [16]. Sukumaran-Rajam and Clauss [21] also detect run-time regularity using linear interpolation and regression models, selecting optimizations in a speculative fashion. However none of this approaches allow to customize the program at compile-time to the specifics of the input sparse matrix, instead generating code that is always-correct whatever the input sparse matrix.

The Sparse Polyhedral Framework [12, 20, 23] provides a unified framework to express affine and irregular parts of the code by representing indirection array access using *uninterpreted function symbols*. In essence this amounts to an (over-)approximation of the non-polyhedral program into a polyhedral one, which is perfect for the purpose of generating automatically at compile-time I/E code. Still, the same advantages and limitations occur: the generated code will be valid for any input sparse matrix, but will not exploit opportunities to customize the optimization to a specific input matrix.

The Trace Reconstruction Engine (TRE) [17] used in this paper uses an algebraic approach to synthesize an affine statement with a single perfectly nested reference which produces a sequence of integers provided as input. The tool works by analyzing the elements in the input sequentially by progressively refining an initial 1-dimensional affine statement, incorporating more elements of the input each time, until a full match is obtained. The tools works by finding all the possible solutions to the linear equation systems which

describe the iteration polyhedron and its projection on the input sequence. The tool can be used to analyze memory address streams, but also sequences of integer indices as we have done in this paper. The TRE has been recently improved to support the synthesis of piecewise-affine domains (i.e., loops using min/max of affine functions in their lower/upper bounds).[4]

Clauss et al. [7] characterized program behavior using polynomial piecewise periodic and linear interpolations separated into adjacent program phases to reduce function complexity. The model can be recursively applied, interpreting coefficients of the periodic interpolation as traces in themselves. Clauss and Kenmei [6] introduced polyhedra to graphically represent the program memory behavior (including cache misses) and facilitate its understanding. Ketterlin and Clauss [11] proposed a method for trace prediction and compression based on representing memory traces as sequences of nested loops with affine bounds and subscripts. Such approach could also be used in place of TRE, but we note that rebuilding multi-statements or their schedule [11] is not needed in our present work.

7 Conclusion

Sparse matrices conveniently store only non-zero values of the matrix. This representation may greatly optimize storage and computation size, but typically prevents polyhedral analysis of the application due to the presence of indirection arrays. This paper introduces our approach to generate polyhedral-friendly code specialized to a given input sparse matrix, by employing affine trace reconstruction algorithms to discover regularity in the set of non-zero coordinates.

Specifically, we focus on immutable sparse matrices, and we illustrate how to build an "affine-SpMV" matrix-vector multiplication operation as a union of dense polyhedral subcomputations. Experimental results highlight the potential for performance improvements but also performance degradation using our technique, as the performance is highly dependent on the number, dimensionality and size of the polyhedral pieces describing sets of non-zero coordinates.

References

[1] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *PLDI*, 1995.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, France, Sept. 2004.

[3] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. *CC*, 6011:283–303, 2010.

[4] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 1991.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, 2008.

[4] This extension is described in an ongoing journal submission

[6] P. Clauss and B. Kenmei. Polyhedral modeling and analysis of memory access profiles. In *Proceedings of the 2006 IEEE International Conference on Application-Specific Systems, Architecture and Processors, ASAP*, pages 191–198, 2006.

[7] P. Clauss, B. Kenmei, and J. C. Beyler. The periodic-linear model of program behavior capture. In *Proceedings of the 11th International Euro-Par Conference*, pages 325–335, 2005.

[8] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *SC*, 1995.

[9] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software*, 38:1–25, 2011.

[10] G. Gupta and S. Rajopadhye. The z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP*, pages 237–248. ACM, 2007.

[11] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th International Symposium on Code Generation and Optimization, CGO*, pages 94–103, 2008.

[12] A. LaMielle and M. Strout. Enabling code gen. with sparse polyhedral framework. Technical report, Colorado State University, 2010.

[13] P. Mucci et al. Performance Application Programming Interface. http://icl.cs.utk.edu/papi/people/index.html. Last accessed November 2017.

[14] R. Ponnusamy, J. H. Saltz, and A. N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *SC*, 1993.

[15] L.-N. Pouchet. the PoCC polyhedral compiler collection. http://pocc.sourceforge.net.

[16] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Distributed memory code generation for mixed irregular/regular computations. In *ACM PPoPP 2015*. ACM, 2015.

[17] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 14th International Symposium on Code Generation and Optimization, CGO*, pages 139–149, 2016.

[18] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 1990.

[19] S. Sharma, R. Ponnusamy, B. Moon, Y. shin Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *SC*, 1994.

[20] M. M. Strout, G. George, and C. Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *LCPC*, September 2012.

[21] A. Sukumaran-Rajam and P. Clauss. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim.*, 12(4):48, 2016.

[22] A. Venkat, M. Hall, and M. Strout. Loop and data transformations for sparse matrix code. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*. Association for Computing Machinery, 2015.

[23] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[24] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[25] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. *LCPC*, 1993.