Generating Piecewise-Regular Code from Irregular Structures

Travis Augustine
Colorado State University, USA
Travis.Augustine@colostate.edu

Janarthanan Sarma
Colorado State University, USA
jsharma@colostate.edu

Louis-Noël Pouchet
Colorado State University, USA
pouchet@colostate.edu

Gabriel Rodríguez
Universidade da Coruña, Spain
gabriel.rodriguez@udc.es

Abstract

Irregular data structures, as exemplified with sparse matrices, have proved to be essential in modern computing. Numerous sparse formats have been investigated to improve the overall performance of Sparse Matrix-Vector multiply (SpMV). But in this work we propose instead to take a fundamentally different approach: to automatically build sets of regular sub-computations by mining for regular sub-regions in the irregular data structure. Our approach leads to code that is specialized to the sparsity structure of the input matrix, but which does not need anymore any indirection array, thereby improving SIMD vectorizability. We particularly focus on small sparse structures (below 10M nonzeros), and demonstrate substantial performance improvements and compaction capabilities compared to a classical CSR implementation and Intel MKL IE's SpMV implementation, evaluating on 200+ different matrices from the SuiteSparse repository.

CCS Concepts • **Software and its engineering → Source code generation**; • **Theory of computation** → *Data compression*; *Program analysis*.

Keywords Polyhedral compilation, SpMV, trace compression, sparse data structure.

ACM Reference Format:
Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating Piecewise-Regular Code from Irregular Structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314615

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314615

1 Introduction

Irregular data structures, as exemplified with sparse matrices, are essential in modern computing: they lie at the core of many applications ranging from physics simulation to graph processing. In machine learning, they help represent sparsely connected neural networks, which themselves can arise after sparsification (weight / connection pruning) of a fully-connected trained network, to improve inference speed.

Numerous sparse formats have already been investigated to improve the overall performance of operations such as Sparse Matrix-Vector multiply (SpMV, e.g. [4, 5, 17, 39]), including focusing on vectorizability of the final program [14, 36]. A compact representation like Compressed Sparse Rows (CSR) [31] ensures only useful (nonzero) values are stored but at the cost of the use of indirection arrays and challenges in effectively SIMD vectorizing the program. In this work we propose instead to take a fundamentally different approach: to automatically build sets of regular sub-computations by mining for regular sub-regions in the irregular data structure. Our approach leads to code that is specialized to the sparsity structure of the input matrix (or tensor), but which does not need anymore any indirection array, thereby improving SIMD vectorizability. This mining is done at compile-time, and requires the sparsity structure (e.g., the nonzero coordinates) to be both known at compile-time and invariant during the computation, as the code generated is valid only for one specific sparsity structure. In this paper, we focus on smaller sparse structures (below 10M nonzeros), and demonstrate substantial performance improvements and compaction capabilities compared to a classical CSR implementation and Intel MKL's SpMV library, evaluating on 200+ different matrices from the SuiteSparse repository.

Our approach targets specifically small-sized sparse-immutable data structures, such as immutable graphs represented as sparse matrices or sparse weight matrices obtained from a fully-connected trained Artificial Neural Networks (ANN) which is sparsified before deployment for inference in production. In a nutshell, we develop algorithms to automatically mine for multidimensional $\mathcal{Z}$-polyhedra [18] that capture sets of nonzero coordinates within the sparse structure, so that each such polyhedron is then a stand-alone subset

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

(a)

```
  for(i = 0; i < n; ++i) {
I: y[i] = 0;
   for(j=0; j<m; ++j)
    if (A_dense[i][j] != 0)
S:   y[i] += A_dense[i][j] * x[j];
  }
```

(b)

```
  for(i = 0; i < 1000; ++i) {
I: y[i] = 0;
   for(j=i; j<=i; ++j)
S:  y[i] += A_data[i] * x[j];
  }
```

(c)

Figure 1. Matrix-vector product implementations. (a) is classical CSR, (b) is classical dense while skipping zero entries, (c) is an example of a simple specialized SpMV for a diagonal matrix encoded in CSR.

of the computation. Benefiting from the associativity/commutativity of the + operation in SpMV, we are able to mine and group together operations from anywhere in the sparse structure, creating more opportunities for code compaction and/or performance. We perform polyhedral code generation to create efficient loop-based code scanning these polyhedra, in turn generating code that not only does not need any indirection array to recover the nonzero coordinates, but can be tuned to favor the exposure of SIMD vectorizable sub-regions when they exist. We make the following contributions:

- a set of algorithms and tools to automatically mine for regular sub-structures in a multidimensional set of integer tuples;
- an end-to-end implementation of a system to generate efficient C code specialized to the computation's sparsity structure;
- an extensive experimental validation over 200+ sparse matrices from the SuiteSparse database as well as a case study on pruning fully-connected ANNs for improved inference speed, demonstrating performance trade-offs and significant gains possible over a classical SpMV CSR implementation and Intel MKL IE.

The paper is organized as follows. Section 2 provides background and overview of the approach. Section 3 presents the core approaches for the automatic extraction of regular pieces from the irregular sparse structure, while Sec. 4 presents hierarchical reconstruction for improved compaction. A case study for ANN sparsification is presented in Sec. 5, before extensive experimental results are detailed in Sec. 6. Related work is discussed in Sec. 7 before concluding.

2 Motivation and Overview

We now outline the problem we address, and the associated challenges in terms of performance and code size compaction. In the following, we illustrate our approach using the SpMV CSR operation while mining for regularity in sparse matrices, but as detailed in the following sections our approach is not limited to this specific computation, and is instead able to handle arbitrary sparse structures as long as each useful entry (nonzero or other) can be represented with a unique integer tuple labeling said entry.

Implementing matrix-vector product We start illustrating our approach by focusing on computing $\vec{y} = A.\vec{x}$, 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 nonzero 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 nonzeros. Figure 1 shows two implementations of this operation: Fig. 1a shows a typical CSR implementation, using `A_data` to store the nonzero data, and `pos` and `cols` arrays to compute/retrieve the coordinates (i, j) associated to a nonzero 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. Fig. 1c illustrates *on an extremely simplified case* what can be the outcome of our system: if the input sparse matrix only stores nonzeros along its diagonal, then the `row` and `cols` array can be entirely removed, the code in Fig. 1c becomes a specialized *regular* matrix-vector product for this sparse matrix structure and offers all the advantages of a dense-only code, including contiguous accesses to all three arrays/vectors `y`, `A_data` and `x`. In a nutshell, our approach is about generating a collection of loop nests that will iterate on only nonzero coordinates, one for each sub-part of the computation where regularity can be discovered and exploited.

Mining a sparse matrix to build polyhedra Our proposed approach intuitively 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 nonzero. We then attempt to rebuild as few polyhedra as possible that capture all and only points in this trace. In that sense, our problem has analogy with affine trace reconstruction [21, 29] 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 nonzero, where $N = M = 1000$. The polyhedron describing the nonzero 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. Then these polyhedra are converted into a loop-based code that scans them, executing the corresponding matrix-vector operation in statement S for each point, this is exactly polyhedral code generation [3]. Figure 1c shows the code specialized to this diagonal matrix, a purely regular/polyhedral program.[1]

Numerous difficulties arise with this approach. First, we must ensure that 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. 1c uses a 1D array for example, which is sufficient to capture in a single domain all nonzero elements here, but as shown below it may also be possible to reduce the number of pieces/polyhedra by increasing their dimensionality. Second, we must control the number and complexity of the polyhedra being rebuilt to describe the sparse matrix. An extreme case where each nonzero is captured in a single polyhedron (one point in it) is always possible, yet would be practically useless.

Trading-off regularity discovery for performance We now illustrate with an actual sparse matrix. Consider the sequence of accesses in Fig. 2a, corresponding to tracing the execution of the SpMV code in Fig. 1a using the matrix `HB/nos1` from the SuiteSparse Matrix Collection [13][2]. A convenient feature of the SpMV computation is that the (i, j) coordinates of each nonzero are 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 nonzeros exist.

As can be seen in Fig. 2b, all the nonzeros lie *nearby* the main diagonal, and zooming on this diagonal in Fig. 2c, we can see upon closer inspection a recognizable sparsity pattern. Our objective is to automatically build a collection of polyhedra $\mathcal{D}_i$ that captures thse sets of nonzero coordinates.

Table 1 displays for the `HB/nos1` the trade-off between the number of pieces (stmts), their maximal dimensionality (i.e., the depth of the loop nest needed to scan a piece), and the performance in cycles obtained by applying a geometric-based approach for reconstructing polyhedra, similar to Rodríguez et al. [28]. Code size is reported as Lines of Code (LoC) in the final reconstructed C program.

Table 1 shows different reconstruction choices, ranging from a single 8D domain (intuitively this will lead to an 8-deep loop nest to scan the polyhedrally compressed matrix) to 312 disjoint 2D pieces. Section 6 discusses the potential performance impacts of such trade-off.

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

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

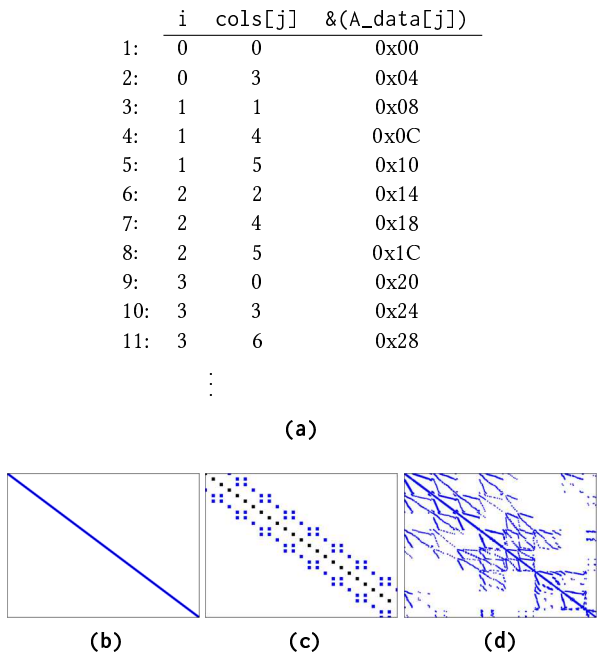

Figure 2. Different sparse matrices from the HB group of the SuiteSparse Matrix Collection. Figure a) shows an excerpt of the accesses performed during SpMV of matrix `HB/nos1`. The nonzero elements in this matrix are shown in Fig. b), and a zoom of its main diagonal is provided in Fig. c). This is a 237×237 matrix with $1,017$ nonzero elements, and its reconstructed code consists of a single statement inside an 8-dimensional loop. Figure 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 reconstructed code includes 870 pieces of up to 8 dimensions.

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).

$\mathbf{max_d}$	2	3	4	5	6	7	8
pieces	312	159	81	4	3	2	1
cycles	11373	11583	9938	35730	34116	39306	50371
LoC	772	1004	671	195	368	165	101

Intuitively, the dimensionality of a piece corresponds to the number of variables used to describe the evolution of values of a set of integer tuples, e.g. the 3D tuple $(i, colsj, addrA)$. For example, the 1D sequence $2, 4, 6, 8$ can be totally captured by a 1D *affine* function $F(i) = 2i : 1 \leq i \leq 4$, but the sequence $2, 4, 8, 10$ cannot.[3] Here a 2D function $F(i, j) = 6i+2j : 0 \leq i \leq 1 \wedge 1 \leq j \leq 2$ instead can capture in a single 2D polyhedron this set of points. Extending the reasoning, there

[3] We only reconstruct affine multidimensional functions, to ensure polyhedral code generation can be applied.

will be typically larger sequences that can be captured by increasing the dimensionality of the pieces. But as shown in Table 1 this may come at a performance cost: indeed, if a dimension is very small (e.g., j in the example above, iterating only twice) we end up branching extremely often in the program just to execute these small loops [28]. The situation is exacerbated with HB/nos1: the entire sparse matrix can be captured with a single 8-dimensional polyhedron, but its shape is so complex that its execution time is 6x slower than when using 470 pieces.

Our approach is therefore as follows. We develop *a collection* of techniques to rebuild polyhedra from *multidimensional* traces, *exploiting the ability to group non-consecutive points in the same polyhedra* (a consequence of exploiting associativity/commutativity on +). These techniques, presented in the next section, include an extension of a geometric approach based on the Trace Reconstruction Engine by Rodríguez et al. [29] and several brute-force pattern-driven mining in the sparse structure to expose easier-to-vectorize micro-codelets, grouped into polyhedra. We further develop hierarchical reconstruction processes, as explained in Sec. 4. As a consequence, the final performance reached for HB/nos1 is down to 5446 cycles, using 423 LoCs, which represents a performance improvement of 1.4× compared to the CSR-based SpMV performance. Details of the setup are provided in Sec. 6. Note that Intel MKL IE specializes in large matrices and incurs a major slowdown here, given that the time for preparing/inspecting the data far exceeds the few thousand cycles needed to compute the actual SpMV operation.

3 From Integer Points to Polyhedra

We now delve into the technical details of our approaches to represent sets of integer tuples with a union of multidimensional polyhedra. We first define concepts and notation, before outlining the algorithms to mine for regularity.

3.1 Concepts and Definitions

We first introduce how elements in an irregular structure are represented in our framework. They must each be uniquely identified by an integer tuple, or point, and the set of all points in the structure are captured with a trace.

Definition 3.1 (Integer tuple). A point $\vec{p}$ is an integer tuple of dimension $dim(p)$. It is noted $\vec{p} = (p_1, ..., p_{dim(p)})$, such that $\forall i,\ p_i \in \mathbb{N}$.

Example 3.2. In Fig. 2a, the point on line 2 is a 3D tuple $\vec{p^2} = (0, 3, 4)$

Definition 3.3 (Trace of points). A trace T is an ordered list of n points $T = \{\vec{p^1}, ..., \vec{p^n}\}$. Its dimensionality is noted $dim(T) = dim(p^i)$, where all points in T necessarily have the same dimensionality. When the trace is reorderable, T can be viewed as a set instead of a list.

Example 3.4. In Fig. 2a, T is the list of all points in the trace, and $dim(T) = 3$.

Note that when the dependences in the computation allow to proceed with grouping points in any order, we call the trace reorderable. This is the case for example with SpMV when exploiting associative reordering. Otherwise we are limited to merging in the same polyhedron only points which appear consecutively in the trace, without the possibility to skip points.

We now define the structures we rebuild from a trace.

Definition 3.5 (affine inequality). Given an integer set of dimension d involving variables noted $(x_1, ..., x_d)$, an affine inequality is written as $\sum_{i=1}^{d} \alpha_i x_i + \beta \geq 0$, where $\forall i, \alpha_i \in \mathbb{Z}$ and $\beta \in \mathbb{Z}$.

Definition 3.6 (integer polyhedron). An integer polyhedron $\mathcal{D}$ of dimension $dim(\mathcal{D})$ contains integer points $\vec{p^i}$, such that $dim(\mathcal{D}) = dim(\vec{p^i})$. It is defined by the intersection of finitely many half-planes defined by affine inequalities. We note a polyhedron by specifying its dimensions name followed by the inequalities defining it: $\mathcal{D} = \{[i_1, ..., i_{dim(\mathcal{D})}] : ineq_1, ..., ineq_p\}$

Example 3.7. In Fig. 2a, considering $p^7 = (2, 4, 24)$ and $p^8 = (2, 5, 28)$. The polyhedron $\mathcal{D}_1 = \{[i, j, k] : i = 2 \wedge 4 \leq j \leq 5 \wedge k = 4j + 8\}$ captures exactly this set of two points. Note that an equality (e.g., $i = 2$) can always be written as a combination of two inequalities (e.g., $i \leq 2 \wedge i \geq 2$).

Definition 3.8 (integer lattice). An integer lattice F is an integer multidimensional function $F : \vec{I} \rightarrow \vec{O}$. F must be represented exactly via a matrix $dim(\vec{O}) \times dim(\vec{I})$ with only integer coefficients. We note a lattice by specifying its input and output dimensions, followed by the equalities defining the function: $F = \{[i_1, ..., i_{dim(\vec{I})}] \rightarrow [o_1, ..., o_{dim(\vec{O})}] : o_1 = f_1(\vec{I}), o_2 = f_2(\vec{I}), ...\}$.

Definition 3.9 ($\mathcal{Z}$-polyhedron). A $\mathcal{Z}$-polyhedron is the intersection of an integer polyhedron and an integer lattice as defined above. Equivalently, the points represented by a $\mathcal{Z}$-polyhedron are the image of $\mathcal{D}$ by F.

Example 3.10. Taking the set of four 1D points 2, 4, 8, 10. The integer lattice $F = \{[i, j] \rightarrow [x] : x = 6i + 2j\}$ and the polyhedron $\mathcal{D}_2 = \{[i, j] : 0 \leq i \leq 1 \wedge 1 \leq j \leq 2\}$ capture exactly this set, which is the image of $\mathcal{D}_2$ by F, that is $F(\mathcal{D}_2)$ takes as values 2,4,8 and 10.

Definition 3.11 ($\mathcal{Z}$-polyhedron origin). The origin of a $\mathcal{Z}$-polyhedron is the lexicographically first point in this $\mathcal{Z}$-polyhedron, i.e., $\vec{p_{\text{orig}}} = lexmin(\mathcal{D} \cap F)$.

Example 3.12. The origin of the $\mathcal{Z}$-polyhedron from Example 3.10 is 2.

Finally, we use standard notation $\cap$, $\cup$ and # for intersection, union, and number of points.

We conclude by defining a reconstructed set of points as a union of $\mathcal{Z}$-polyhedra of varying dimensions. This also corresponds to a polyhedral representation [18, 26] of the irregular structure.

Definition 3.13 (Reconstructed trace). A reconstructed trace T_{poly} from the trace T of n elements is a list of finitely many $\mathcal{Z}$-polyhedra D_i noted $T_{poly} = \{D_1, ..., D_p\}$, such that $\sum_i \#D_i = n$ and $T \equiv T_{poly}$ (i.e., T_{poly} captures the same exact set of points as T). Any two $\mathcal{Z}$-polyhedra D_i and D_j in T_{poly} may have different dimensionalities and different number of points.

Example 3.14. Returning to Fig. 2a. Considering the trace to be reorderable, we can group non-contiguous points together in the same polyhedron. Consider $p^6 = (2, 2, 20)$ and $p^{10} = (3, 3, 36)$. The polyhedron $\mathcal{D}_1 = \{[i, j, k] : 2 \leq i \leq 3 \wedge i = j \wedge k = 16i - 12\}$ models these two points. $\mathcal{D}_2 = \{[i, j, k] : 0 \leq i \leq 1 \wedge i = j \wedge k = 8i\}$ models p^1 and p^3. $\mathcal{D}_3 = \{[i, j, k] : 1 \leq i \leq 2 \wedge 4 \leq j \leq 5 \wedge k = 12i + 4j - 16\}$ models p^4, p^5, p^7 and p^8. Taking $p^2 = (0, 3, 4)$ and $p^{11} = (3, 6, 40)$, the image of the polyhedron $\mathcal{D}_4 = \{[i, j, k] : 0 \leq i \leq 1 \wedge j = 3i + 3 \wedge k = 36i + 4\}$ by the lattice $F_4 = \{[i, j, k] \rightarrow [x, y, z] : x = 3i \wedge y = j \wedge z = k\}$ models exactly p^2 and p^{11}. Finally $\mathcal{D}_5 = \{[i, j, k] : i = 3 \wedge j = 0 \wedge k = 32\}$ models the single point p^9. For all these polyhedra except $\mathcal{D}_4$, the identity lattice $F = \{[i, j, k] \rightarrow [x, y, z] : i = x \wedge j = y \wedge k = z\}$ is applied to form the $\mathcal{Z}$-polyhedra $D_1 = \mathcal{D}_1 \cap F$, $D_2 = \mathcal{D}_2 \cap F$, etc. The complete reconstructed trace is therefore the collection D_1, D_2, D_3, D_4, D_5 which exactly describes the 11 points in the original trace. Observe some polyhedra may contain only a single point, a worst-case solution which is always possible for any trace, but does not achieve compression. Finally, we remark this is only one possible reconstruction out of many possible ones, Table 1 displays various possible reconstructions of the full trace from which this example is extracted. Loop-based code scanning each of these polyhedra is generated from the vertices of the polyhedra, using a simplified CLooG [3]-like algorithm. Intuitively, a loop is created for every dimension of the polyhedron, iterating on the range of values it can take. In this work we ensure the vertices and stride of the polyhedra reconstructed are explicit, for all reconstruction algorithms, so code generation is straightforward.

3.2 Geometric Trace Compression

The first method we present is an extension of the original Trace Reconstruction Engine (TRE) approach by Rodríguez et al. [29] which is targeted at reconstructing into polyhedra a stream of addresses, as typically originating from memory accesses when executing the program. In contrast to their work [29], we have both different objectives and different requirements: we have as objective to reconstruct a multidimensional stream (i.e., points $\vec{p^i}$ where $dim(\vec{p^i}) \geq 1$, while TRE is limited to $dim(\vec{p^i}) = 1$) which complicates the problem; but we do not need to preserve the order of points in T_{poly} if the trace is reorderable as is the case for SpMV, a great additional degree of freedom.

The reader may refer to [29, 30] for complete details about the TRE algorithm, which we only slightly extend here. In a nutshell, our extended TRE rebuilds a $\mathcal{Z}$-polyhedron that captures a multidimensional 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. 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 $(\mathcal{D}_A, F_{i_1}, F_{i_2}, ..., F_{i_n})$ to be reconstructed, for a trace of dimension n. We achieve this by modifying TRE to instead analyze n streams simultaneously, with the additional constraint that $\mathcal{D}$ is identical for all n streams, i.e., only the reconstructed F_{i_j} can be different across different streams. Precisely for SpMV, three streams correspond to building $F_{A_{data}}$, F_i and F_j each being a stream of scalar values. For each nonzero 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 $F_{A_{data}}$ also, which captures the memory location of the data associated to a particular (i, j) coordinate, *the polyhedral generated code can operate by indexing directly the input sparse matrix representation containing the nonzero elements (including, but not limited to, CSR).* This means data does not need to be copied into 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.

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 [29]. For this reason, timeout mechanisms are included to halt the analysis when it does not achieve significant advances after a specified number of steps.

Algorithm 1 shows a simplified pseudocode of key steps of our extended TRE (eTRE). Essentially, the processing starts with an empty $\mathcal{Z}$-polyhedron, and tries to enlarge it by sequentially adding points from the trace. A list of candidate polyhedra 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 and increase its dimensionality to incorporate one new point to the polyhedron, and continue processing it.

Algorithm 1: Pseudocode of eTRE

```
Input: Trace T
Output: List of Z-polyhedra
Function addIters( T, D)
    p⃗ = lexicographicMaximum(D);
    p⃗_next = lexicographicNext(p⃗);
    if ExistsInTrace(p⃗_next) then
        D = D ∪ p⃗_next;
        D = D ∪ addIters(T, D);
    end
    else
        return D;
    end
end
Function TRE( T)
    L = emptyList;
    while True do
        D = retrieveBestZPoly(L);
        if timed-out or T = ∅ then
            removePointsFromTrace(T, D);
            return { D } ∪ TRE(T)
        end
        D = increaseDimensionality(D);
        L = appendList(L, addIters(T, D));
    end
end
```

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 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, as illustrated in Sec. 2. But, as also illustrated in Sec. 2 and previously studied by Rodríguez et al. [28], performance does not necessarily correlate with the reconstruction size.

3.3 Pattern-Matching to Mine for Codelets

The eTRE approach depicted above provides the ability in the general case to rebuild polyhedra from a trace. But it remains driven by geometric criteria that include a window-based approach: it consumes points consecutively in the trace, trying to add them in a polyhedron, bailing and creating a new piece when it is not possible. Furthermore, for the sake of compaction, points could be added leading to impossible-to-vectorize loops (e.g., non-constant stride in the inner-most loop) or having very poor locality (e.g., consecutively executed points being far away in memory).

To overcome this limitation, we present a pattern-matching based approach, to be used in complement with eTRE. This approach only works for reorderable traces, as it is driven by the ability to group points in a polyhedron that can appear anywhere in the trace, including non-consecutive points. It is then to be combined with eTRE, as described in Sec. 4.

Micro-codelets The starting point is to define a family of template shapes we mine for, that is, partially defined $\mathcal{Z}$-polyhedra of specific shapes aimed in part at improving SIMD vectorization opportunities. We present below templates for the 3D integer tuples that model traces for SpMV. It is straightforward to generalize to lower or higher dimensionality. Our approach can seamlessly handle sparse tensors of arbitrary dimensionality, as exemplified in Sec. 5.

Specifically, we are mining for hyperrectangles[4] of any size within a given range, allowing strides between points. The prototype polyhedron shape we are mining for is therefore:

$$\mathcal{D}_{3Dcodelet} = \{[i,j,k] : m_i \leq i \leq M_i \wedge m_j \leq j \leq M_j \wedge m_k \leq k \leq M_k\}$$

and the prototype lattice is:

$$F_{3Dcodelet} = \{[i,j,k] \rightarrow [i',j',k'] : i' = s_i i \wedge j' = s_j j \wedge k' = s_k k\}$$

The unknowns, which are found by mining the actual trace, are $m_i, M_i, m_j, M_j, m_k, M_k \in \mathbb{N}$ which define the 8 vertices of the polyhedron; and $s_i, s_j, s_k \in \mathbb{N}$ which define the constant stride allowed between points in the $\mathcal{Z}$-polyhedron. Note that the polyhedron origin is therefore by construction $\vec{p_{orig}} = (m_i, m_j, m_k)$.

The algorithm Equipped with this template capturing all rectangles with constant strides, we proceed by mining the trace from the largest to the smallest rectangle size to find all places where the template can be applied to model points in the trace. Algorithm 2 describes this process. Note that our implementation is in practice slightly different to improve the exploration speed, but it produces the same outcome. In essence, the algorithm searches from the largest to the smallest rectangle in terms of the number of points, searching from the smallest to largest stride (i.e., distance between points). Function ZpolyhedronFromVertices builds a $\mathcal{Z}$-polyhedron from its vertices and stride by instantiating the values from the template defined above. Finally, note the high complexity of this algorithm: it is important to control the maximal size of the rectangle and maximal stride explored tightly. In practice, we search for rectangles containing at most $5 \times 5 \times 5$ points, with a maximal stride of 5, and avoiding rectangles containing less than two points.

For example, a rectangle of size $4 \times 1 \times 1$ with a stride of 1 between points along the first dimension is a set of consecutive operations along the i dimension, which can be SIMD-vectorized trivially. Our implementation favors finding vectorizable rectangles first (i.e., those whose size will be

[4] A hyperrectangle is a n-dimensional generalization of a rectangle. For simplicity, we use the term rectangle when referring to a hyperrectangle.

multiple of the vector length along at least one dimension), that is in practice the `for` loops iterating on the rectangle sizes and stride follow a more specific order than simply decreasing the size continuously, but the same set of possibilities as in Alg. 2 is explored.

Algorithm 2: Micro-codelet mining

```
Input: Trace T, Msz_i, Msz_j, Msz_k the maximal polyhedron sizes,
       Mst_i, Mst_j, Mst_k the maximal strides
Output: List of Z-polyhedra
Function CodeletMiner(T,Msz_i, Msz_j, Msz_k, Mst_i, Mst_j, Mst_k)
    L = emptyList;
    T_reord = lexicographicSort(T);
    for s_i in [Msz_i..1], s_j in [Msz_j..1], s_k in [Msz_k..1],st_i in
      [1..Mst_i], st_j in [1..Mst_j], st_k in [1..Mst_k] do
        p⃗ = firstPointInTrace(T_reord);
        m_i = p_1, m_j = p_2, m_k = p_3;
        M_i = m_i + s_i, M_j = m_j + s_j, M_k = m_k + s_k;
        S = ∅;
        invalid = false;
        for i in [m_i..M_i], j in [m_j..M_j], k in [m_k..M_k] do
            p⃗_searched = (i * st_i, j * st_j, k * st_k);
            if ExistsInTrace(p⃗_searched, T_reord) then
                S = S ∪ p⃗_searched;
            else
                invalid = true;
                break;
            end
        end
        if invalid = false then
            T_reord = removePointsFromTrace(T_reord,S);
            L = append(L, ZpolyhedronFromVertices(m_i, M_i, m_j, M_j,
              m_k, M_k, st_i, st_j, st_k));
        end
    end
    return L;
end
```

4 From Polyhedra to Compact and Efficient Code

The previous section describes how to reconstruct a trace as polyhedra, using either a multidimensional geometric approach (eTRE), or a codelet-centric approach. In both cases, a fully functional program can already be reconstructed, by generating polyhedron-scanning code for each $\mathcal{Z}$-polyhedra (e.g., using CLooG [3]), turning them into loop-based code scanning each point in them.

We now describe how to combine both approaches, by performing hierarchical reconstruction.

4.1 Mining for Macro-Codelets

The first natural hierarchical composition approach is to find ways to combine micro-codelets together, that is, attempting to find regularity *in the micro-codelet origins*. Our approach is straightforward, in that it is in essence a recursive call to Alg. 2, depicted in Algorithm 3 below. We refer to the polyhedron built from micro-codelets as a macro-codelet.

We simply call recursively Alg. 2 on the subtrace made only of the $\mathcal{Z}$-polyhedra origins, and proceed by typically extending the maximal stride constraint to improve the chances of grouping micro-codelets together. In essence, we are attempting to build a polyhedron of polyhedra, each uniquely identified by its origin. To ensure easy code generation we restrict to only grouping polyhedra of identical shape and stride, in turn the final polyhedron is built simply by combining the dimensions and constraints of the micro-codelets considered with the macro-codelet polyhedron.

Algorithm 3: Macro-codelet mining

```
Input: Trace T, micro-codelet maximal sizes, macro-codelet maximal sizes
Output: List of Z-polyhedra
Function HierarchicalMiner(T, micro- and macro-codelet max. sizes)
    L = emptyList;
    L_bottom = CodeletMiner(T, micro-codelet max. sizes);
    for each unique shape size s in L_bottom do
        T^s_origins = buildTraceFromAllPolyOrigins(L,s);
        L_s = CodeletMiner(T^s_origins, macro-codelet max. sizes);
        L = append(L,L_s);
    end
    return L;
end
```

This algorithm can be naturally extended in a recursive fashion. However we have found typically very few if any opportunity in finding polyhedra of polyhedra of polyhedra when reconstructing sparse matrices for example. In this work, we have limited to the two-level reconstruction depicted in Alg. 3.

4.2 Geometric Compression of Micro-codelets

The second hierarchical reconstruction approach we propose is combining the power of the eTRE geometric reconstruction, which can lead to non-rectangular shapes of any dimensionality, with micro-codelet origins. The algorithm is essentially the same as Alg. 3 except CodeletMiner is substituted by eTRE: it follows the same approach of building new traces made of only micro-codelet origins and operating on such sub-traces.

We remark that, in practice, we attempt to first build macro-codelets. For the micro-codelet origins which have not been grouped in polyhedra containing at least four points, we then resort to using eTRE to attempt to group these left-over micro-codelets. The rationale is that eTRE will typically search for significantly more complex shapes than hyper-rectangles: it will search for arbitrary orientations of the faces, as well as searching for polyhedra of increasing dimensionality. It therefore can find compaction opportunities the macro-codelet approach cannot discover, but possibly requiring complex loop control to scan the obtained polyhedra. In practice, we limit the maximal dimensionality of polyhedra rebuilt by eTRE on top of micro-codelets to 3, as experiments confirmed no benefit in terms of performance in attempting higher dimensionalities, as already suggested by Rodríguez et al. [28].

4.3 Efficient Synthesis for Micro-codelets

In order to achieve good performance, the picture must be completed by observing an inherent drawback of our approach: as we mine for regularity in an irregular structure, we are tributary from the existence of such regularity. When only very small micro-codelets can be found, this includes micro-codelets of two or one points, and therefore the number of instructions in the generated program can be proportional to the trace size. As Rodríguez et al. observed, performance is hurt in particular because of instruction cache misses. To address this limitation and ensure good performance even in cases where little regularity is found, we post-process the final ASM code generated to insert prefetching instructions *for the program code*.

In order to alleviate the effect of instruction cache misses for matrices with a working set (code plus data) approximately fitting the last-level cache, we introduce aggressive instruction prefetching into our executables. For this purpose, we coded a tool to analyze the assembly code of the executed SpMV kernels, and introduce explicit calls to `prefetcht1`, targeting blocks of code to be executed in the future. This transformation is outlined in Algorithm 4. In addition, codes shall be stored in 2 MB memory pages in order to minimize the number and impact of iTLB misses.

Algorithm 4: Pseudocode of the prefetch insertion

```
   Input: Assembly code of a function f
   Output: Modified assembly code with periodical prefetches
1  Function add_prefetch(f)
2      offset = 0;
3      foreach instruction i in f do
4          if offset >= 64 bytes and not_in_loop(i) then
5              insert prefetcht1 before i;
6              offset = sizeof_prefetch_instruction;
7          end
8          offset += sizeof_instruction(i);
9      end
10 end
```

Note that built-in prefetch calls were not designed for prefetching code. For this reason, neither `prefetcht0` nor `prefetchnta`, which bring blocks to the data L1, are appropriate, as these blocks will not be found when fetching instructions. We choose instead to bring them to the unified L2, from where the blocks will be fetched upon a miss in I1. The prefetch distance is heuristically chosen to be 4096 bytes, as this provided the best performance improvement. We conjecture that, given that prefetch hints do not need to be followed by the processor, using a smaller prefetch distance does not give enough time to actually find a relatively idle moment to emit latent prefetches. We insert one prefetch after each 64 bytes of executable code, i.e., we prefetch every single block in the code after the first 4096 bytes. This increases total code size by approximately 10%.

5 Case Study: Efficient Inference of Pruned Neural Networks

We now present a case study to illustrate the potential of our approach beyond already-existing sparse matrices. Section 6 extensively reports the performance of our approaches for efficient execution of SpMV on 200+ matrices. We focus here on another source for sparsity: pruning a trained neural network to speed up inference.

5.1 Pruning Fully-Connected Artificial Neural Networks

For this case study, we focus on two small fully-connected ANNs which have been trained to recognize digits on the MNIST dataset [24]. These networks take as input a 28×28 gray-scale image, and output one probability for each of the 10 possible digits (0 to 9).

Two networks for digit recognition The first, noted 1L, is a simple network with an input layer of 784 elements directly connected to the output layer with 10, for a total of 7,840 base operations. The second, noted 3L, features the same input/output layer as 1L, but has two hidden layers made of 21 and 20 neurons respectively, for a total of 16,330 base operations. These two trained networks have been provided to us by colleagues expert in machine learning. Our objective here is to speed up the inference execution time.

The trained networks are represented using dense weight matrices, as a fully-connected network architecture was trained for, as is typically done. These matrices are of size 784×10 for 1L, and for 3L we have three weight matrices: one of 784×20, another of 20×21, and the last of size 20×10. Inference proceeds typically layer-by-layer, computing a matrix-vector operation between the weight matrix and the vector representing the values of the previous layer (or the input layer). In between each layer the hyperbolic tangent is computed to threshold the values, and a traditional softmax is implemented on the output layer. The 1L network achieves 93% accuracy (9,255 correctly predicted / 745 mispredicted), while 3L achieves 96% accuracy (9,630/370); both are respectable values in terms of prediction quality.

Network pruning While it is customary to train fully-connected networks especially for such problem of digit classification, it is also well-known that a possibly much smaller network may be trained while achieving similar prediction accuracy. Techniques such as pruning neurons/connections which do not contribute much e.g. [20] or re-training a simpler network e.g. [19] have been developed to reduce the footprint of the network and improving its inference speed. The challenge in turn is to execute effectively a SpMV operation, as after pruning the weight matrices may become significantly sparse.

Our objective for this case study is to perform a pruning of 1L and 3L that is solely driven by the importance/contribution of each weight, ignoring any consideration such as vectorizability or easiness to reconstruct the code. That is, we sparsify the weight matrices solely based on the individual contribution of each individual weight. Our algorithm for pruning proceeds as follows. We first compute a pixel contribution table, that for each of the 784 pixels quantifies its significance towards a particular output. We then, for each of the 10 possible outputs, use this table to order all operations in the inference of the network contributing to this output by their relative contribution to the final result. We discard the hyperbolic tangent during this analysis to avoid thresholding effects when computing combined contributions, but it is restored in the final code produced. In essence, the output of this pruning algorithm is a trace of operations indexed by the layer, source and target neuron coordinates, which we then reconstruct into polyhedra.

We do not claim there is a specific contribution in our pruning strategy, it was developed only to illustrate the ability of our techniques to reconstruct "arbitrary schedules" applied to an input set of operations. Table 2 shows the accuracy we obtain when executing only the first 30% and first 50% of operations, from the list of all operations ordered by their relative contribution. Therefore the pruning ratios are 70% and 50% for these two cases. We also compare against a simple random approach to remove weights and another somewhat simple approach of removing weights that exceed a threshold, to ensure our pruning strategy is actually meaningful. For all entries, inference over the entire 10,000 images in MNIST has been performed.

Table 2. Pruning accuracy (% correctly predicted images)

	full	random		threshold		reorder-by-sign.	
		30%	50%	30%	50%	30%	50%
1L	93%	52%	74%	84%	90%	**85%**	**90%**
3L	96%	15%	18%	90%	95%	**90%**	**96%**

5.2 Reconstruction Results

We applied the techniques presented in previous sections to reconstruct an efficient code from the sparsified trace of the first 30% and first 50% points after reordering by significance. The performance is summarized in Table 3. The baseline is a fully-regular loop nest for dense matrix-vector product that computes inference on the full networks, without pruning.

Table 3. Performance obtained, in cycles

	baseline	30%	50%
1L	36565	9587	12415
3L	69914	16972	25587

Table 3 presents the final execution time *after pruning*, that is, executing only 30% or 50% of the operations that the baseline computes. As shown in Table 2, this still leads to nearly identical accuracy. We get a double benefit: reducing significantly the number of operations, and executing the remaining ones very fast, by exposing micro-codelets which are very effectively vectorized by Intel ICC. This experiment indicates an interesting potential for our approach, which we reserve for future work: develop a pruning algorithm that is aware of the reconstruction objectives for performance, to ensure we implement inference using essentially only micro-codelets, for best vector performance.

6 Experimental Results

6.1 Experimental Setup

The polyhedral reconstruction process has been applied to the sparse matrix – vector multiplication kernel to evaluate its potential for linear algebra computations. For this purpose, the full SuiteSparse collection [13] is targeted, limited to matrices with at most 10 million nonzero elements for tractability purposes. This yields 2,637 matrices for which a polyhedral reconstruction of their SpMV kernel was built. Afterwards, a sieve of these results was performed to reduce the experimental set while retaining the representativity of the full matrix suite. The selection process is detailed in Figure 3.

Experiments were executed on an Intel Core i7 8700K with 64 GB of RAM memory. The CPU frequency was fixed at the base frequency of 3.7 GHz to prevent thermal constraints affecting experimental variability. Transparent hugepages of 2 MB are automatically used to store the data segment of all codes. The polyhedral C codes implementing SpMV for each selected matrix were automatically synthesized and compiled using ICC v18.0.3 with `-O2 -xSKYLAKE -vec-threshold0`, to ensure that the vectorization capabilities of the machine were fully employed and that all vectorizable operations were executed as such, regardless of expected profitability. We experimentally verified that using `-O3` did not only increase compilation times noticeably, but resulted in slightly underperforming programs.

6.2 Experimental Results

Figure 4 shows the speedup obtained by our reconstructed codes with respect to the best baseline version (either the classical irregular SpMV code, or Intel MKL, whichever is faster). Note that this data does not use the prefetch insertion described in Sec. 4.3. The impact of prefetch insertion and the use of hugepages for code memory is discussed later below.

As can be observed, the initially good performance degrades as the number of nonzeros increases. In order to study this effect, we extracted and analyzed counters related to execution performance and memory behavior. An excerpt

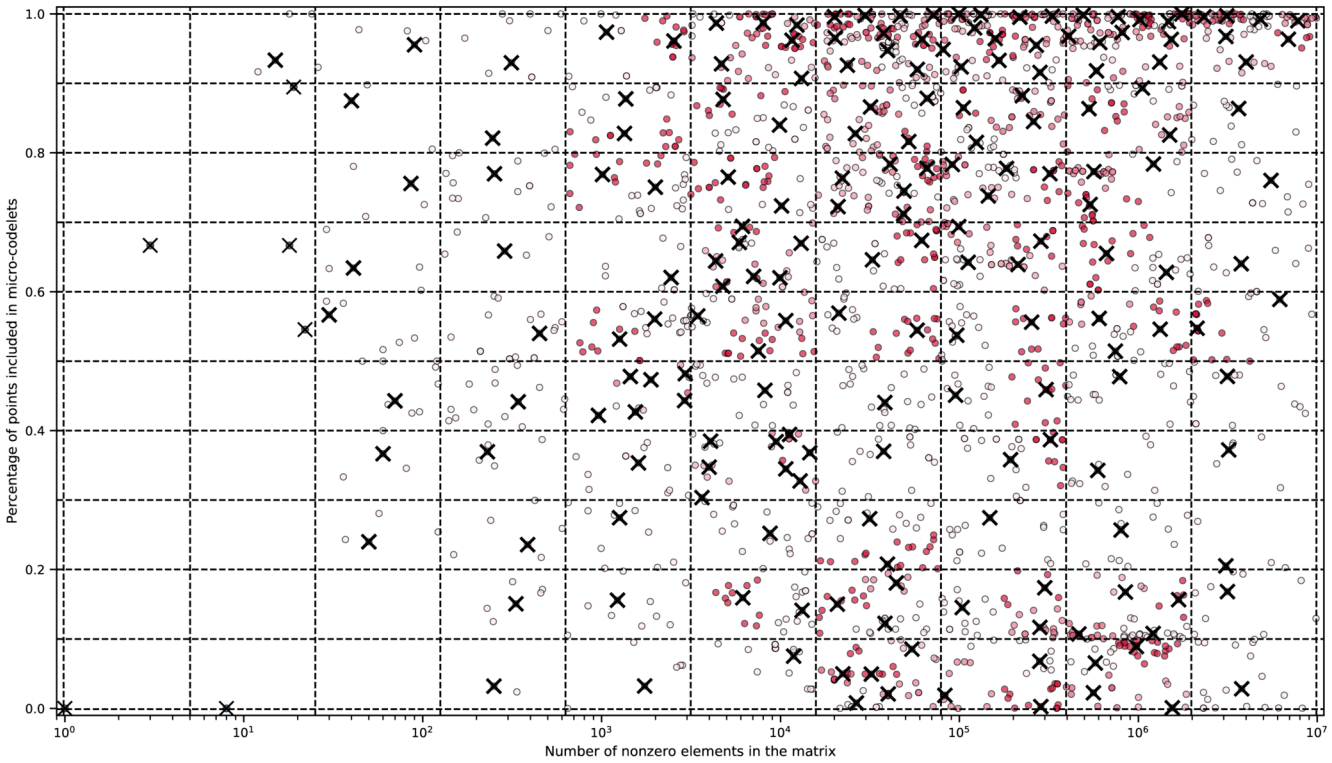

Figure 3. Sieve of the 2,637 SuiteSparse matrices below 10 million nonzeros. First, the SpMV kernel for each of these matrices is reconstructed and analyzed. These are classified according to the decile they belong to in terms of matrix size (logarithmically) and percentage of points included in micro-codelets. That yields one hundred buckets, as shown in the figure. Afterwards, inside each bucket, a k-means clustering process is run to select representative matrices. The number of clusters k for each bucket is selected to be representative of the probability density of matrices in that bucket, but each non-empty bucket will have at least one matrix selected. The different clusters created by the process are shown in different colors in the figure. Afterwards, the matrices closest to each cluster center are selected for the final test set, and marked with an 'X' in the figure. In total, 200 matrices were selected.

of these counters for particularly relevant matrices is shown in Table 4. As can be seen, matrices which exhibit good performance, such as `Newman/power`, are characterized by a drastic reduction in the number of executed instructions, stemming from the elimination of loops; and, critically, by a good last-level cache behavior. But the size of the executable increases with the number of nonzeros in the sparse matrix. Eventually, the executable gets too large, and the instruction misses in the last-level cache dominate the execution time, as is the case for matrix `FIDAP/ex19`. To address this issue we introduce instruction prefetching as outlined in Sec. 4.3. Figure 5 shows the performance improvements by using this technique. Large executables also lead to a degraded performance due to iTLB misses. To address this issue, we store the text segment of polyhedral codes in 2 MB hugepages. This optimization has a performance impact of up to 20% for the largest matrices in the experimental set. We found no positive impact to storing the text segment of irregular SpMV and Intel MKL codes in hugepages.

Using fully unrolled codes composed of micro-codelets only and prefetching is not enough to bring performance advantages for very large codes, where the working set size is larger than the last-level cache. In these cases, as seen in the performance counters of the SpMV of matrix `GHS_indef/sparsine` in Table 4, the pollution of the last-level cache by blocks of code negatively impacts the memory performance of the kernel, and removes the advantages brought by vectorization. In order to reduce code sizes, we perform hierarchical reconstructions as described in Section 4. Note that hierarchical reconstruction is not applicable to all matrices, as it depends on exploiting regularity in the shape of the matrix that might not be there. Furthermore, increasing code dimensionality has diminishing returns. The first limitation is of a structural nature: the vectorizable sections of code which are recognized in a first pass are relatively homogeneous, as there are limited computation shapes in this step. However, when applying hierarchical reconstruction on top of these shapes the results are much more varied

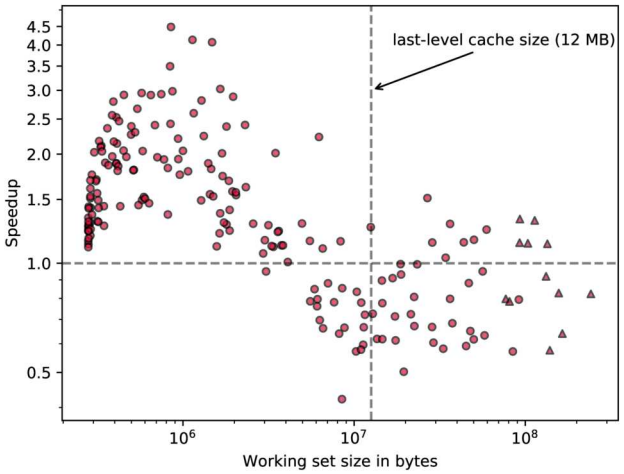

Figure 4. Speedup of the reconstructed polyhedral version, with no prefetches added, with respect to the best performing baseline version. Circles indicate that the best performing version was the classical irregular SpMV code, while triangles correspond to matrices for which the best performing baseline is Intel IE MKL. As shown in the figure, MKL is only relevant for the largest matrices in the experimental set.

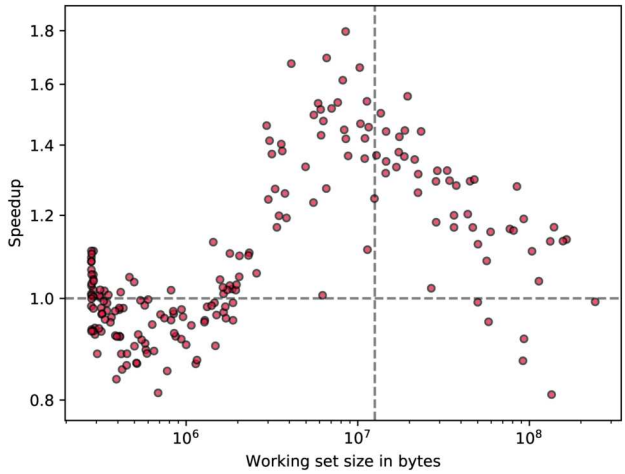

Figure 5. Speedup of the prefetched version with respect to the original, non-prefetched one. Code prefetches help take advantage of cache size by overlapping computation and code fetches from memory. The technique loses effectiveness for small codes and for working set sizes above the size of the last-level cache.

Table 4. Performance counters for selected matrices.

	group matrix	Newman power	FIDAP ex19	GHS_indef sparsine
NNZ		13188	259577	1548988
cycles	irregular	168534	739414	7064292
	poly. nopf	37563	1752757	11169230
	poly. pf	40552	975626	11760144
inst. count	irregular	211323	1611527	8710349
	poly. nopf	37284	682220	4372122
	poly. pf	42295	755473	4936644
mem. access.	irregular	69380	795183	4593163
	poly. nopf	34289	356122	3697497
	poly. pf	39300	429375	4262018
D1m	irregular	1061	9157	862083
	poly. nopf	1890	18046	918569
	poly. pf	2316	26815	850316
I1m	irregular	79	90	110
	poly. nopf	4858	71685	536917
	poly. pf	5391	78716	605120
L2m	irregular	7571	95461	1216023
	poly. nopf	8942	118846	1571362
	poly. pf	11372	158351	2011163
L2im	irregular	65	73	73
	poly. nopf	4751	70468	535984
	poly. pf	144	1664	8719
L3m	irregular	0	23	355478
	poly. nopf	0	338	1225380
	poly. pf	0	13843	771530

and heterogeneous. As a result, finding identically-shaped higher dimensional macro-codelets to fuse becomes a rarer occurrence. The second limitation is practical: we have experimentally discovered that, starting with 5-dimensional loops, induction variables cannot be held in CPU registers and they have to be accessed through the cache, heavily degrading performance. For this reason, and considering that our base codes have no loops (vectorizable micro-codelets are issued as vector operations, while non-vectorizable fragments are executed in isolation), we decide to limit the reconstruction steps to three. In each of them, identical sections of the reconstructed polyhedra are fused together into a higher dimensionality polyhedron, thus achieving code compression.

While hierarchical reconstruction does not necessarily improve performance, it does achieve code size reduction for most large matrices. For matrices which expose regularity this allows to dramatically reduce code sizes, at the cost of increasing the number of loop-related instructions and therefore the total instruction count. Performance improvements may stem from two different sources: i) reducing the code size imposes less pressure on the cache hierarchy, improving memory behavior; and ii) increasing loop depth may improve vectorization efficiency. On the opposite end, hierarchically reconstructed codes execute more total instructions due to the control of additional loops; and may even worsen memory behavior depending on which micro-codelets are executed within a macro-codelet, and in which order.

Figure 6 shows selected performance counters for the five matrices which achieve the most compression to exemplify interesting tradeoffs from the hierarchical reconstruction process. The TSOPF matrix is an example where the percentage of operations executed with a vector length of 4 is greatly increased. As a result, performance improves by 12% for the 2-dimensional code. However, the 3-dimensional code is 15% slower than the baseline, due to a 30% instruction count increase and no further advantages in terms of memory behavior. The matrix which achieves the best speedup

is GHS_psdef/apache2, with up to 55% speedup in the 2-dimensional version. Vector operations increase significantly, but in this case the better memory behavior is the culprit of the improvement. On the side of matrices which do not benefit at all from hierarchical reconstruction from the performance point of view is Zhao/Zhao2, which does not find new vectorization opportunities, nor does improve memory behavior, leading to a 15% slowdown. Note how the instruction count increases in most cases. The only notable exceptions are those where the improvement provided by an increase in vector operations offsets the increase in control flow instructions.

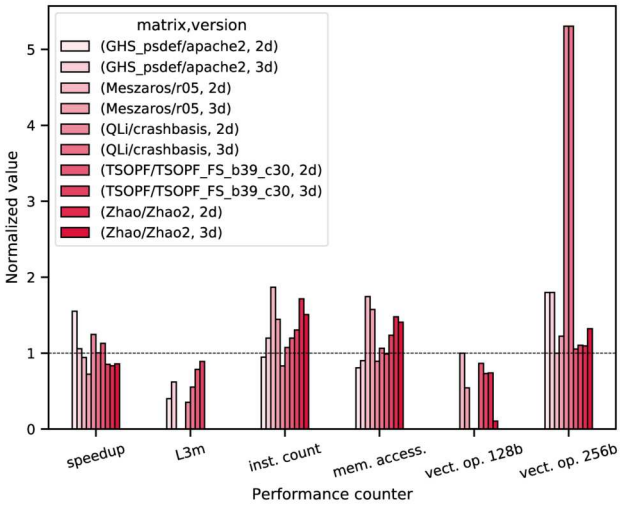

Figure 6. Selected performance counters for the five matrices which achieve a better compression from hierarchical reconstruction. Values are normalized to those obtained by the plain polyhedral reconstruction without loops.

We present four figures to summarize the global performance/compression aspects of the polyhedral hierarchical reconstruction of SpMV kernels. Figures 7 and 8 show the best speedups and compressions, respectively, achieved by polyhedral reconstructions of different dimensionalities. Figure 9 shows the performance, in terms of cycles per FLOP, for the SpMV of matrices larger than 100 nonzero elements. As illustrated by this figure and Figure 7, the polyhedral reconstruction achieves a sweet spot in between 1,000 and approximately 100,000 nonzeros, with performance improvements up to 4x.

Figure 10 shows how the generated code size relates to the number of nonzeros in the matrix. The number of Lines of Code (LoC) in the reconstructed program is bounded by the number of nonzero in the matrix, as in the worst case one instruction per nonzero will be generated. Note there is also a small number of LoCs always added for preprocessor directives and function headers.

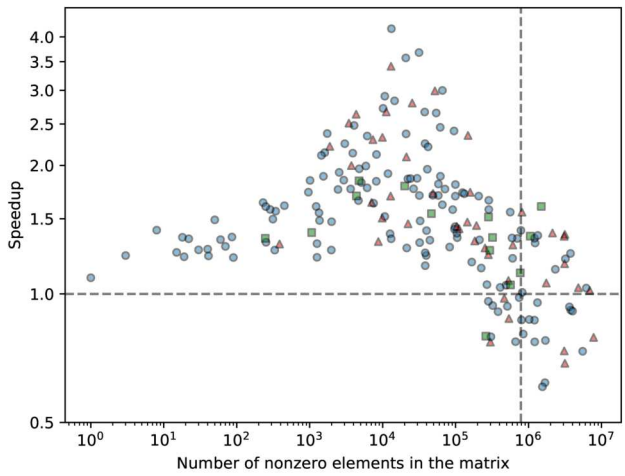

Figure 7. Best speedup obtained for polyhedral codes with respect to the best performing baseline. The vertical dashed line marks the approximate size of the last-level cache. Different markers are used depending on the best polyhedral version: a circle for micro-codelets-only codes, a triangle for micro-codelets grouped under single-level (1D) loops, and a square for codelets grouped under 2D loop nests.

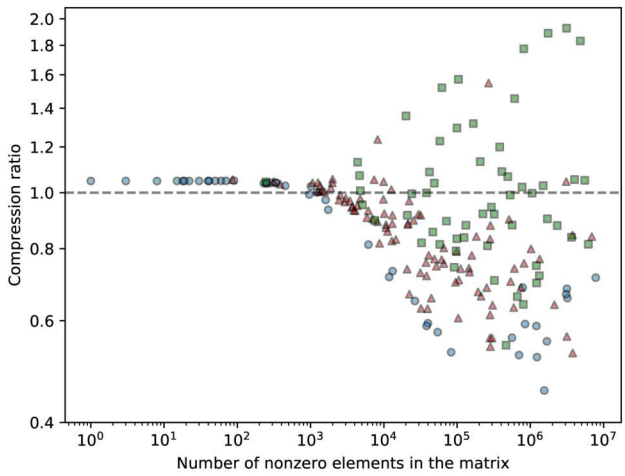

Figure 8. Best compression obtained for polyhedral codes with respect to the best performing baseline. Markers (circle, triangle, square) have the same semantics as in Figure 7. Results are obtained compiling with -O2 and automatic prefetch insertion, which improves performance by 12% and increases code sizes by 30%, on average, with respect to compiling with -Os and without prefetching.

Synthesis time The complete process to synthesize the reconstructed program is made of three stages. (1) Generating the trace to obtain the set of nonzero coordinates, which typically takes between a few milliseconds and a few seconds for the biggest matrices of millions of nonzeros we

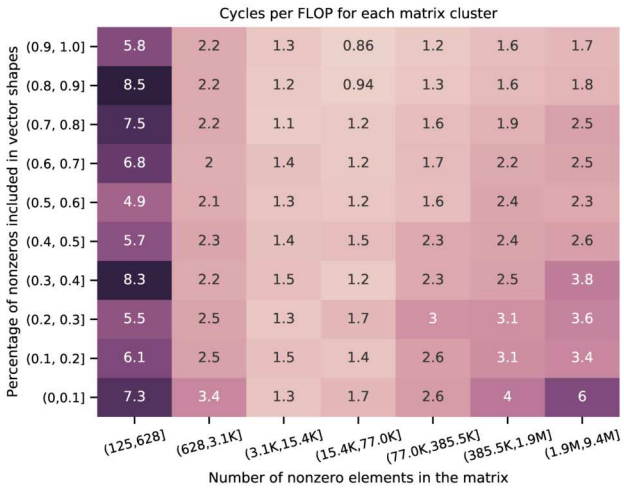

Figure 9. Heatmap of the cycles per FLOP in categories of matrices above 100 nonzeros. Best results for the different hierarchical reconstructions are reported.

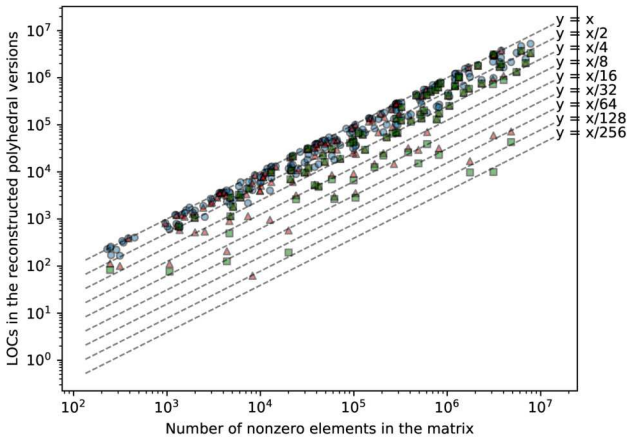

Figure 10. Code sizes of polyhedral reconstructions for matrices above 100 nonzeros. Markers (circle, triangle, square) have the same semantics as in Figure 7. Dashed lines mark the ratio between LOCs generated and the number of nonzeros in the sparse matrix.

report on. (2) Mining for regularity by using the algorithms presented above, which typically takes anywhere between a few hundred milliseconds and a few minutes for complex traces. Note that the number of nonzeros is very often a good predictor of the reconstruction time, with a correlation factor $R = 0.94$ between both. However the reconstruction time complexity here is not only a function of the trace size, but also of the sparsity pattern, leading to some outliers taking significantly more time. The longest ever reconstruction time we observed was 80 minutes in our experiments for `Rucci/Rucci1` which contains 7.8M nonzeros. (3) Performing polyhedral code generation and emit the final C program, which is typically very fast (one minute or less). As the vertices of the micro-codelets are known at reconstruction time we use a simplified version of the CLooG [3] code generator algorithm.

7 Related Work

Optimizing sparse vector-matrix multiply has mostly been investigated from two fronts: (a) improving the sparse representation itself; and (b) inspecting at run-time the sparsity to pack data for an executor program. None of these approaches achieve the same objective as ours, that is to create a polyhedral (piecewise-regular) representation at compile-time of the irregular structure for subsequent code generation and optimization.

Sparse formats There is an extensive amount of prior work on customizing sparse matrix formats and optimizing SpMV on different platforms. This is only related work, in that to the best of our knowledge our work is the first to automatically mine for regularity in sparse data structures by rebuilding a polyhedral representation for regular sub-pieces, in turn enabling polyhedral code generation of the sparsity structure. We also recall that we focus on small sparse structures (below 10M nonzeros), which are typically much less investigated in general SpMV work that mostly focus on executing very large sparse matrices efficiently. Intel MKL IE is a good example: the overhead of just initiating the library is prohibitive for small matrices.

Some formats require additional tuning of many architecture- or kernel-specific parameters in order to achieve best performance. For example, Vuduc [39] presented an automated system for generating efficient implementations of SpMV on CPUs, while Williams et al. [40] moved toward multi-core platforms with the implementation of parallel SpMV kernels. For GPUs, Bell and Garland have implemented sparse matrix formats in CUDA [5] and proposed the HYB approach (hybrid of ELL and COO). Choi et al. [8] introduced the concept of blocks for CSR and ELL (i.e. BCSR and BELLPACK). Block-based formats present similarities in objective with our approach, by ensuring the sparse matrix is viewed as a collection of regular blocks of matrix coordinates. However, blocks represent a contiguous set of coordinates, which often need to include storage for zeros and typically must have the same size across the entire matrix. Our approach does not have any of these restrictions. Yang et al. [42] studied the use of blocks for matrices that present large graphs with power-low characteristics, combining Transposed Jagged Diagonal Storage (TJDS) [15] with COO and blocking. Note that GPU implementations of these formats are logically significantly impacted by how nonzeros are distributed across thread-blocks and threads [2, 4, 5, 8, 16, 41, 42].

The TACO compiler [9, 22] is a framework for generating code for optimized (sparse) tensor algebra computations,

with advanced code synthesis capabilities supporting a variety of sparse formats. In particular, it emits code to efficiently compute on tensors stored in any combination of formats, via clever abstractions about sparse formats. Integrating our work as a custom sparse format based on $\mathcal{Z}$-polyhedra inside the TACO compiler would be very interesting, as it has the potential to deliver additional performance especially when the sparse tensors operated on are sparse-immutable.

Inspector/executor methods 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. [32]. They developed runtime infrastructures for distributed memory parallelization of irregular applications [25, 32, 33]. These were augmented with compiler approaches that automatically generated parallel code [1, 12, 38]. Ravishankar et al. [27] exploit runtime regularity to produce polyhedrally-optimizable executor code in specific cases. Sukumaran-Rajam and Clauss [35] also detect runtime 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 [23, 34, 37] 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 program for a specific matrix.

Cheshmi et al. [6, 7] developed Sympiler, an I/E compiler to optimize sparse computations by exploiting properties of the sparsity structure. This leads to executor code which leverages the sparsity pattern within the computation. In contrast to our work, indirection arrays are not fully eliminated in the final generated code, appearing both in the access functions and in the loop bounds.

Polyhedral trace compression The Trace Reconstruction Engine (TRE) [29] built upon 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. It finds 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. TRE has been improved to also support the synthesis of piecewise-affine domains (i.e., loops using min/max of affine functions in their lower/upper bounds) [28].

Clauss et al. [11] 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 [10] introduced polyhedra to graphically represent the program memory behavior (including cache misses) and facilitate its understanding. Ketterlin and Clauss [21] 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 [21] is not needed in our present work.

8 Conclusion

Numerous sparse formats have already been investigated to improve the overall performance of Sparse Matrix-Vector multiply (SpMV). In this work we took a radically different approach: synthesize code that is specialized to a particular sparse structure, automatically building sets of regular sub-computations by mining for regular sub-regions in the irregular data structure. We then perform polyhedral code generation to create efficient loop-based code scanning these polyhedra, in turn generating code that not only does not need any indirection array to recover the nonzero coordinates, but can be tuned to favor the exposure of SIMD blocks.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation award CCF-1750399, and by the Ministry of Economy, Industry and Competitiveness of Spain (TIN2016-75845-P AEI/FEDER/EU).

References

[1] G. Agrawal, J. Saltz, and R. Das. 1995. Interprocedural Partial Redundancy Elimination and its Application to Distributed Memory Compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. La Jolla, CA, USA, 258–269.

[2] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. 2014. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. New Orleans, LA, USA, 781–792.

[3] C. Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *13th International Conference on Parallel Architectures and Compilation Techniques, PACT*. IEEE, Antibes, France, 7–16.

[4] N. Bell and M. Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[5] N. Bell and M. Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *ACM/IEEE Conference on High Performance Computing, SC*. Portland, OR, USA.

[6] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 13.

[7] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. 2018. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 62.

[8] J.W. Choi, A. Singh, and R.W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. Bangalore, India, 115–126.

[9] S. Chou, F. Kjolstad, and S. Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 123.

[10] P. Clauss and B. Kenmei. 2006. Polyhedral Modeling and Analysis of Memory Access Profiles. In *IEEE International Conference on Application-Specific Systems, Architecture and Processors, ASAP*. Steamboat Springs, CO, USA, 191–198.

[11] P. Clauss, B. Kenmei, and J. C. Beyler. 2005. The Periodic-Linear Model of Program Behavior Capture. In *11th International Euro-Par Conference*. Lisbon, Portugal, 325–335.

[12] R. Das, P. Havlak, J. Saltz, and K. Kennedy. 1995. Index Array Flattening Through Program Transformation. In *ACM/IEEE Supercomputing Conference, SC*. San Diego, CA, USA, Article 70.

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

[14] E.F. D'Azevedo, M.R. Fahey, and R.T. Mills. 2005. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *Intl. Conference on Computational Science, ICCS*. Atlanta, GA, USA, 99–106.

[15] A. Ekambaram and E. Montagne. 2003. An Alternative Compressed Storage Format for Sparse Matrices. In *Intl. Symposium on Computer Science and Information Sciences, ISCIS*. Antalya, Turkey, 196–203.

[16] J. Godwin, J. Holewinski, and P. Sadayappan. 2012. High-performance Sparse Matrix-vector Multiplication on GPUs for Structured Grid Computations. In *5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU*. London, UK, 47–56.

[17] R.G. Grimes, D.R. Kincaid, and D.M. Young. 1980. *ITPACK 2.0: User's Guide*. http://books.google.com/books?id=h8RcNAAACAAJ

[18] G. Gupta and S. Rajopadhye. 2007. The Z-Polyhedral Model. In *12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. San Jose, CA, USA, 237–248.

[19] S. Han, J. Pool, J. Tran, and W. Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems, NIPS*. Quebec, Canada, 1135–1143.

[20] B. Hassibi and D.G. Stork. 1992. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In *Advances in Neural Information Processing Systems, NIPS*. Denver, CO, USA, 164–171.

[21] A. Ketterlin and P. Clauss. 2008. Prediction and Trace Compression of Data Access Addresses through Nested Loop Recognition. In *6th International Symposium on Code Generation and Optimization, CGO*. Boston, MA, USA, 94–103.

[22] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 77.

[23] A. LaMielle and M. Strout. 2010. *Enabling Code Generation within the Sparse Polyhedral Framework*. Technical Report. Colorado State University.

[24] Y. LeCun, C. Cortes, and C. Burges. [n. d.]. The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/. Last accessed: April 2019.

[25] R. Ponnusamy, J.H. Saltz, and A.N. Choudhary. 1993. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *ACM/IEEE Conference on Supercomputing, SC*. Portland, OR, USA, 361–370.

[26] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *Proc. Symposium on Principles of Programming Languages (POPL '11)*. ACM, 549–562.

[27] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. ACM, San Francisco, CA, USA, 65–75.

[28] G. Rodríguez and L.-N. Pouchet. 2018. Polyhedral Modeling of Immutable Sparse Matrices. In *8th International Workshop on Polyhedral Compilation Techniques*. Manchester, UK.

[29] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. 2016. Trace-based Affine Reconstruction of Codes. In *Proceedings of the 14th International Symposium on Code Generation and Optimization, CGO*. Barcelona, Spain, 139–149.

[30] G. Rodríguez, M. T. Kandemir, and J. Touriño. 2018. Affine Modeling of Program Traces. *ACM. Trans. Comput.* 68, 2 (2018), 294–300.

[31] Y. Saad. 1990. SPARSKIT: A basic tool kit for sparse matrix computations. (1990).

[32] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. 1990. Run-time Scheduling and Execution of Loops on Message Passing Machines. *J. Parallel Distrib. Comput.* 8, 4 (1990), 303–312.

[33] S. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. 1994. Run-time and Compile-time Support for Adaptive Irregular Problems. In *ACM/IEEE Conference on Supercomputing, SC*. Washington, DC, USA, 97–106.

[34] M.M. Strout, G. George, and C. Olschanowsky. 2012. Set and Relation Manipulation for the Sparse Polyhedral Framework. In *25th International Workshop on Languages and Compilers for Parallel Computing, LCPC*. Tokyo, Japan, 61–75.

[35] A. Sukumaran-Rajam and P. Clauss. 2016. The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12, 4 (2016), 48.

[36] W.T. Tang, R. Zhao, M. Lu, Y. Liang, H.P. Huynh, X. Li, and R.S.M. Goh. 2015. Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi. In *13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO*. IEEE Computer Society, San Francisco, CA, USA, 136–145.

[37] A. Venkat, M.S. Mohammadi, J. Park, H. Rong, R. Barik, M.M. Strout, and M. Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. Salt Lake City, UT, USA, Article 41.

[38] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. 1992. Compiler analysis for irregular problems in Fortran D. In *6th International Workshop on Languages and Compilers for Parallel Computing, LCPC*. New Haven, CT, USA, 97–111.

[39] R.W. Vuduc. 2004. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph.D. Dissertation. University of California.

[40] S. Williams, L. Oliker, R.W. Vuduc, J. Shalf, K.A. Yelick, and J. Demmel. 2009. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. *Parallel Comput.* 35, 3 (2009), 178–194.

[41] S. Yan, C. Li, Y. Zhang, and H. Zhou. 2014. yaSpMV: Yet Another SpMV Framework on GPUs. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. ACM, Orlando, FL, USA, 107–118.

[42] X. Yang, S. Parthasarathy, and P. Sadayappan. 2011. Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining. *Proc. VLDB Endow.* 4, 4 (2011), 231–242.