Automatic Specialization of Polyhedral Programs on Sparse Structures

Alec Sadler
Laboratoire de l'Informatique du Parallélisme
Inria, CNRS, ENS de Lyon, UCBL
Lyon, France
Firstname.Lastname@inria.fr

Christophe Alias
Laboratoire de l'Informatique du Parallélisme
Inria, CNRS, ENS de Lyon, UCBL
Lyon, France
Firstname.Lastname@inria.fr

Abstract

Most High-Performance Computing applications manipulate sparse data, which results in highly irregular code whose compile-time optimization is quite challenging. One way is to start from the original dense specification, which is usually much more regular and ready to be optimized thanks to state-of-the-art program optimization algorithms. This paper presents an algorithm to specialize a dense polyhedral program on sparse inputs. Our approach is able to propagate the input sparse structure across the computation and to keep only the necessary computations computing non-zero values. It is meant to be coupled with other code transformation strategies to generate efficient parallel code. The key ingredient of our algorithm is the transitive closure of affine relations, for which efficient and accurate heuristics exist. Experimental evaluation assesses the scalability and the accuracy of our approach.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Just-in-time compilers**; • **Theory of computation** → **Program analysis**; **Regular languages**; • **Computing methodologies** → *Distributed computing methodologies.*

Keywords: Code optimization, Sparse computation, Runtime specialization

1 Introduction

Since the early days of parallel computing, industry is pushing towards programming models, languages and compilers to help the programmer in the tedious task to parallelize a program. Automatic parallelization focuses on programming automatically parallel computers from a sequential specification. In the past decades, a general unified framework, the *polyhedral model* [19], was designed to solve that problem for *regular* loop kernels manipulating *dense* tensors (arrays). With the polyhedral model, compilers may reason about programs at iteration-level, giving rise to powerful automatic parallelization algorithms [2, 10, 28].

However, most kernels of interest in high-performance computing manipulate *sparse* tensors which can also appear in pruned machine learning models. Sparse codes are *highly irregular* and make use of array indirections and dynamic control which jeopardize static automatic parallelization algorithms. Hence, alternative directions are investigated, such as *runtime parallelization* of sparse code and *compile-time code generation* from a dense code specification when the sparse input data are available.

Runtime sparse parallelizers [37, 39, 40] parallelize the sparse code during the execution by relying on an inspector/executor scheme. The *inspector* retrieves the dependences at runtime from the sparse data and makes the parallelization decisions. Then, the *executor* executes the parallelized code. *Compile-time sparse code generators* on the other hand produce an optimized sparse code from a dense code specification with annotations. Since the seminal work of Bik [6] on efficient sparse data structure selection, several approaches were proposed to produce a sparse code with a minimal amount of computations. Kjolstad [26] proposes a source-to-source code generator, TACO, exploiting the properties of the integer ring $(\mathbb{Z}, +, \times)$ (no multiplication by 0) and generalized to any pool of user defined operators over integers with a set of axiomatic properties [20]. Some approaches [12, 47] have shown the importance of decoupling the symbolic analysis phase from the latter numerical computation stage. The symbolic phase apply code transformations by reasoning over the input sparse structures, and then give the hand to the numerical phase to perform the computation. While these approaches target a multitude of kernels, it still is a sub-set of SCoP programs which the polyhedral model manipulates.

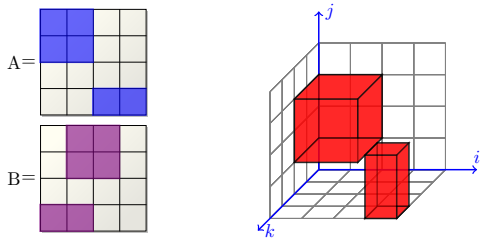
In this paper, we propose a general method for the automatic specialization of dense polyhedral programs on sparse input tensors. We introduce the concept of sparse propagation in the program, where only iterations in loops that generate a non-zero value are kept in the final code. Our approach has many applications in sparse code optimization. It could be used at compile time coupled with various symbolic phase techniques to produce a versionning on different sparse matrix shapes; or at runtime with inspector/executor methods to specialize dynamically a code once the sparse structure is known, and then enable deeper optimizations than those taking the sparse code without any knowledge of the original dense code. Specifically, we make the following contributions:

IMPACT, 2025, Barcelona, Spain

.

```
C[i][j] = reduce(+,(i,j,k->i,j),
                 1 <= i,j,k <= N,
                 A[i][k]*B[k][j]);
```

(a) Dense kernel



(b) Sparse specialization

Figure 1. Motivating example 1: reduction

- We propose an algorithm for specializing (dense) polyhedral programs given a sparse data description. Our approach is based on the direct resolution of fixpoint equations using an abstraction to a formal language.
- We have evaluated our approach on large sparse matrices. The results show that our method is accurate and that the bottlenecks are easily parallelizable.

This paper is structured as follows. Section 3 introduces the polyhedral model and the mathematical tools used in the algorithm description. Section 4 presents the related work. Section 5 presents our specialization algorithm. Section 6 presents our experimental validation. Finally, Section 7 concludes this paper and outlines research perspectives.

2 Motivating Examples

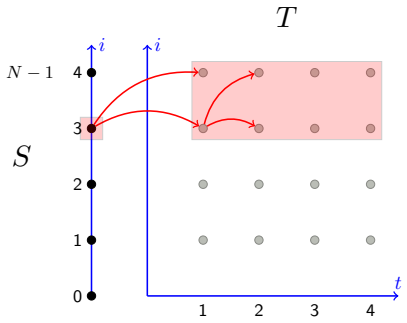
This section presents the motivating examples, which will be discussed throughout this paper. We first consider an example with a *reduction*, an operator widely used in HPC. For the sake of completeness, non-reduction *recurrences* must be handled. They are addressed through a second example.

Example 1: reduction. Consider the matrix multiplication kernel depicted on Figure 1.(a). It is expressed as a single reduction $C[i,j] = \sum_{k=1}^{N} A[i,k] * B[k,j]$. Given input sparse matrices, our goal is to propagate the sparsity across the computation, and keep only the useful computations depicted in (b). For example here, iterations (i,j,k) are kept only if $A[i,k] \neq 0$ and $B[k,j] \neq 0$, and finally $C[i,j] \neq 0$ only if there exists some k with $A[i,k] \neq 0$ and $B[k,j] \neq 0$. On our example, we would only have relevant loop iterations depicted in red on the picture.

Example 2: recurrence. Consider the kernel depicted on Figure 2.(a). Again, we want to keep all iterations which produces a non-zero element after propagating the sparse

```
  for(i = 0; i <= N-1; i++)
S:   A[0,i] = In[i];
  for (t = 1; t <= M; t++)
    for (i = 1; i <= N-1; i++)
T:    A[t,i] = A[t-1,i-1] + A[t-1,i];
```

(a) Dense kernel



(b) Sparse specialization

Figure 2. Motivating example 2: recurrence

input In. On T, we would keep the iterations (t,i) satisfying:

$$A[t-1,i-1] \neq 0 \text{ or } A[t-1,i] \neq 0$$

Somehow, we need to check recursively $A[t-1,i-1]$ and $A[t-1,i]$ until a non-zero input $In[i']$ is reached. This can become very hard to compute as the number of dependences grow in the program, it is also not trivial when input In also need to be computed from other previous kernels.

The outcome of our algorithm is the set of loop iterations producing a non-zero value for each statement of the program. For instance, this could be used to generate a specialized program focusing on relevant iterations thanks to state-of-the-art polyhedral code generation tools [4].

3 Preliminaries

This section oulines the fundamental notions behind our algorithm. Section 3.1 presents the polyhedral model, the general compilation framework in which our work fits. Then, Section 3.2 presents our intermediate representation, the systems of affine recurrence equations. Finally, Section 3.3 presents affine relations, the main mathematical tool used in our approach.

3.1 Polyhedral model

The polyhedral model [16–19, 33, 34] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [19] and data locality improvement [10]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [16],

scheduling [18] or loop tiling [10] to quote a few). The polyhedral model manipulates program fragments consisting of nested `for` loops and conditionals manipulating arrays and scalar variables, such that loop bounds, conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters (input sizes, e.g., N). Thus, the control is static and may be analysed at compile-time. With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$ and is called an *operation* or an *execution instance*. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S.

Example (cont'd). Both examples are clearly a polyhedral program. Example 2 has two statements S and T, whose iteration domains are *polyhedra parametrized by and M and N*: $\mathcal{D}_S = \{i \mid 0 \leq i \leq N-1\}$, $\mathcal{D}_T = \{(t,i) \mid 1 \leq t \leq M, 1 \leq i \leq N-1\}$.

3.2 Systems of Affine Recurrence Equations

Usually, polyhedral programs are not manipulated directly but through an intermediate dataflow representation called a system of affine recurrence equations (SARE) which focuses on the computation itself and abstracts away the storage allocation and the execution order [16]. In a nutshell, a SARE is a dynamic single assignment form manipulating arrays. This means that each array cell is defined (e.g. written once) by a recurrence involving other arrays.

Example (cont'd).

- *Example 1.* The SARE is:

$$C[i,j] \quad = \quad \sum_{(i,j)=\Pi(i,j,k), 1 \leq i,j,k \leq N} A[i,k] * B[k,j]$$

  Π is defined as $\Pi(i,j,k) = (i,j)$. This means that the result $C[i,j]$ is obtained by reducing all the iterations $(i,j,k) \in D$.

- *Example 2.* The SARE is:

$$\begin{array}{rcl} S[i] & = & \texttt{if } 0 \leq i \leq N-1 \texttt{ then } In[i] \\ & & \texttt{else } \bot \\ T[t,i] & = & \texttt{if } (t,i) \in \mathcal{D}_T \texttt{ then} \\ & & \quad (\texttt{if } t = 1 \text{ or } i = 1 \texttt{ then } S[i-1] \\ & & \quad\ \texttt{else } T[t-1,i-1]) + \\ & & \quad (\texttt{if } t = 1 \texttt{ then } S[i] \\ & & \quad\ \texttt{else } T[t-1,i]) \\ & & \texttt{else } \bot \end{array}$$

  To simplify the presentation, the constraints $1 \leq t \leq M, 1 \leq i \leq N-1$ are written $(t,i) \in \mathcal{D}_T$.

There are several equivalent ways to define a SARE. In this paper, we consider the arrays as infinite objects where meaningless cells are filled with the undefined symbol $\bot$. Note that $\bot$ is absorbing for any operator. Also, we will consider arrays as mappings assigning a value to each index of $\mathbb{Z}^d$. Arrays will be defined with *functional equations* by means of a *composition of functions* (array reindexing, conditions as functions, constants as functions, etc):

$$\begin{array}{rcl} \mathit{Sare} & ::= & \mathit{Def}+ \\ \mathit{Def} & ::= & \mathit{Array} := i \mapsto \mathit{Expr} \\ \mathit{Expr} & ::= & \mathit{Constant} \mid \mathit{Array}[\phi(i)] \\ & & \mid \texttt{if } \mathit{Cond} \texttt{ then } \mathit{Expr} \texttt{ else } \mathit{Expr} \\ & & \mid \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} * \mathit{Expr} \mid \bot \\ & & \mid \sum_{i=\Pi(j), j \in D} \mathit{Expr} \end{array}$$

Figure 3. SARE syntax

- *Example 1.*

$$C \quad = \quad (i,j) \mapsto \textstyle\sum_{(i,j)=\Pi(i,j,k), 1 \leq i,j,k \leq N} A[i,k] * B[k,j]$$

- *Example 2.*

$$\begin{array}{rcl} S & := & i \mapsto \texttt{if } 0 \leq i \leq N-1 \texttt{ then } In[i] \\ & & \texttt{else } \bot \\ T & := & (t,i) \mapsto \texttt{if } (t,i) \in \mathcal{D}_T \texttt{ then} \\ & & \quad (\texttt{if } t = 1 \text{ or } i = 1 \texttt{ then } S[i-1] \\ & & \quad\ \texttt{else } T[t-1,i-1]) + \\ & & \quad (\texttt{if } t = 1 \texttt{ then } S[i] \\ & & \quad\ \texttt{else } T[t-1,i]) \\ & & \texttt{else } \bot \end{array}$$

Syntax. The SARE syntax is depicted on Figure 3, where *Constant* denotes a constant value, *Array* denotes an array identifier, ϕ denotes an affine function and *Cond* denotes an affine condition. We assume the SARE to be well formed. The affine expressions defining ϕ and *Cond* should only involve the indices of the written left hand side array and the structure parameters (e.g. N). The *reduction* is instanciated with a + operator, but it could be any associative/commutative operator with the neutral element 0. Also, the SARE is assumed to be computable – here, it is always the case since it is derived from a polyhedral program and not considered as a standalone input.

Semantics. The SARE semantics is depicted on Figure 4. The semantics of a SARE array M is a mapping $\mathcal{S}[\![M]\!]$ which assigns to each array index the value stored at that index. Each subexpression e is naturally viewed in the same way, this enable the definition of $\mathcal{S}[\![M]\!]$ with functional equations. For instance, the expression $\mathit{Array}[\phi(i)]$ is view as the remapped array $i \mapsto \mathit{Array}[\phi(i)]$. Our semantic relies on a straightforward expression semantics $\mathcal{E}[\![.]\!]$ and condition semantics $\mathcal{B}[\![.]\!]$. Note that the parameter values are implicit, $x : k \mapsto x_k$ only contains the binding for each indice value. Finally, the set of values taken by array cells is enriched with the undefined symbol $\bot$, the same symbol used in the syntax definition for the sake of clarty.

$$\mathcal{S}[\![Array]\!](x) = \mathcal{S}[\![Expr]\!](x) \text{ if } Array := i \mapsto Expr \text{ and } x \in \mathbb{Z}^{\dim i} \tag{1}$$

$$\mathcal{S}[\![Constant]\!](x) = \mathcal{E}[\![Constant]\!] \tag{2}$$

$$\mathcal{S}[\![Array[\phi(i)]]\!](x) = \mathcal{S}[\![Array]\!](\phi(x)) \tag{3}$$

$$\mathcal{S}[\![\texttt{if } Cond \texttt{ then } Expr_1 \texttt{ else } Expr_2]\!](x) = \mathcal{S}[\![Expr_1]\!](x) \text{ if } \mathcal{B}[\![Cond]\!](x) \text{ is true, } \mathcal{S}[\![Expr_2]\!](x) \text{ otherwise} \tag{4}$$

$$\mathcal{S}[\![Expr_1 + Expr_2]\!](x) = \mathcal{S}[\![Expr_1]\!](x) + \mathcal{S}[\![Expr_2]\!](x) \tag{5}$$

$$\mathcal{S}[\![Expr_1 * Expr_2]\!](x) = \mathcal{S}[\![Expr_1]\!](x)\mathcal{S}[\![Expr_2]\!](x) \tag{6}$$

$$\mathcal{S}[\![\bot]\!](x) = \bot \tag{7}$$

$$\mathcal{S}[\![\sum_{i=\Pi(j), j\in D} Expr]\!](x) = \sum_{x=\Pi(j), j\in D} \mathcal{S}[\![Expr]\!](j) \tag{8}$$

Figure 4. SARE semantics

3.3 Affine Relations

We now present affine relations, the main mathematical tool used in our approach. A *Presburger relation* is a binary relation $\rightarrow \subseteq \mathbb{Z}^n \times \mathbb{Z}^p$ such that there exists a Presburger formula Φ (arithmetic over $(\mathbb{Z}, +)$) with $n + p$ free variables with $(x_1, \dots x_n) \rightarrow (y_1, \dots y_p)$ iff x, y satisfy Φ: $(x_1, \dots, x_n, y_1, \dots, y_p) \vdash \Phi$. Usually, those relations are referred to as *affine relations*, as they manipulate affine forms.
Example (cont'd). The following affine relation:

$$\{(i, k, N) \rightarrow (i, j, k, N) \mid 1 \leq i, j, k \leq N\}$$

relates $A[i, k]$ to the iterations reading it. Usually, parameters (N) are omitted to simplify the formulation: $\{(i, k) \rightarrow (i, j, k) \mid 1 \leq i, j, k \leq N\}$.

In this paper, we will use extensively affine relations and particularly their transitive closure (to emulate loops) $\rightarrow^* = \bigcup_{k\in\mathbb{N}} \rightarrow^k$. In general transitive closures of affine relations are not computable, as multiplication could be emulated, and this way the (indecidable) Peano arithmetic. However, pattern matching-based heuristics appears to be very efficient and effective in practice [9, 44]. In the following, we will use the following notation for relation composition: $R_1.R_2$ means $R_2 \circ R_1$. For homogeneity purpose, we will also view a Presburger set P as a the affine relation: $\{() \rightarrow x \mid x \in P\}$. For instance, the set of iterations reading $A[0][0]$ could simply be obtained with $\{() \rightarrow (0, 0)\}.\{(i, k) \rightarrow (i, j, k) \mid 1 \leq i, j, k \leq N\}$.

4 Related Work

In this section, we first outline the work around the generation and the optimization of sparse code; then we outline the work on code specialization. As explained in the paper, many work were done for efficient support of specialization and code generation of sparse code, which are both essential for performance, as code generated by sparse compilers can greatly vary depending on the operation and the data structure of the input tensors.

Sparse code generators. Specification over irregular structures has been tackled by numerous compilers. The MT1 compiler [7, 8] first introduced compilation techniques to transform dense layout into sparse iteration, such as guard encapsulation to iterate over only nonzero elements in a dimension. The Bernouilli project [32] studied a relational approach on sparse computation, linking iteration spaces of diferent tensors as query expressions. These transformations were later refined by the influential TACO compiler [25], a library and code generator. TACO thinks of tensors as tree, with each level corresponding to a dimension (defined as dense or compressed/sparse). When generating code, TACO looks at the overall computation and generates loop nests where each level become a for or a while loops depending on the structure of the many sparse tensors at that level. While efficient, this representation can have trouble expressing iteration over exotic that can iterate over multiple dimensions or blocked layouts [14], which can prove to be more efficient on modern GPUs [5, 13, 36].

Sparse code optimizers. Sparse code optimizers are able to perform code transformation, such as tiling and loop coalescing on sparse code. The polyhedral model[19] originally targets loop nests with regular loop bounds in order to automatically find parallelism, but de facto cannot perform transformations on sparse irregular accesses. Works were done extending the model on sparse computation, as Augustine et al [3] used trace reconstruction in order to exploit patterns for the sparse matrix vector multiplication kernel. This approach was tested with polyhedral code generation tools, but would only generate code of great size with many loops of small trip count that would crash for the biggest matrices from the SparseSuite collection. Zhao et al. [46] compute static upper bound and model control dependences of non-affine loop bounds. The Sparse Polyhedral Framework [38] combine the polyhedral model with inspector-executor methods to target non-affine loops and introduced the concept of uninterpreted function in the polyhedral model to

allow sparse code generation. How those functions are deduced is part of what this paper aims to solve, and can very well be coupled to generate efficient sparse code for programs supported by polyhedral parsers. Venkat et al.[42] in the SPF describe loop transformations to compose sparse layout to polyhedral scanning. Zhao et al. [47] implement co-iteration in the SPF by iterating over one sparse tensor and looking up the indices of the other layouts throught *find* algorithms deducted by an SMT solver. Sparso [35] and COMET [41] describe data reordering techniques to improve locality during computation. Looplet [1], based on same iteration model as TACO, offers a solution to support a variety of underlying structures in sparse tensors, allowing better iteration strategies over dimensions. Overall many of these techniques can benefit from our method for final code generation.

Code specialization. Specialization on sparse data structure is closely intertwined with partial evaluation [15], in which program inputs are split into *static* (for example sparse layouts) and *dynamic* (loop bounds) parts. This allow compilers to generate efficient code which can be adapted to (ie static) components, which is crucial knowing that many sparse layouts [27] still prove to be efficient in essential computations, which then need multiple implementations. Augustine et al.[3] was later refined by many techniques[11, 21, 45] to provide efficient vectorization of SpMV kernels with specialization on the input sparse structure. We aim to generalize these techniques to other kernels.

5 Our Approach

Figure 5 depicts the main steps of our approach. First, we compute a set of *sparsity equations* whose solution is exactly the regions of interest for each tensor used by the program (Section 5.1). There is no direct resolution method, hence we propose to rephrase these equations as *rational language equations* for which resolution methods exists (Section 5.2). This rephrasing will *abstract* away the difficult parts. Finally, we translate back the solution as a *simplified* system of sparsity equations (Section 5.3). If it is directly solvable, we compute the solution (1). If not, the system might need to be *evaluated*, or *interpreted*, to get the final solution (2). This is discussed in Section 5.4.

5.1 Compiling the Sparsity Equations

First, we generate a set of *sparsity equations*, whose solution gives the regions of interest for each tensor manipulated by the program. The equations are generated by the syntax-directed translation rules depicted on Figure 6. Given a SARE array S, $[\![S]\!]$ denotes the *set* of non-zero indices of S. The rules start from the non-zero indices of input arrays $[\![I_k]\!]$ and then propagate the non-zero elements across the SARE arrays using the rule $0x = x0 = 0$ (rule 15), and assuming that $x + y \neq 0$ whenever $x \neq 0$ or $y \neq 0$ (rule 13), which is, of course, an overapproximation as $x + (-x) = 0$ for $x \neq 0$. The

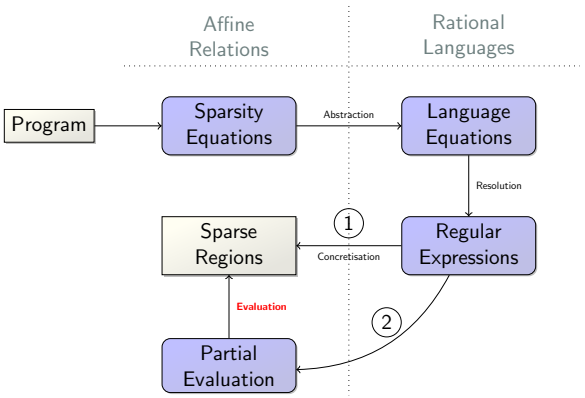


Figure 5. Overview

subtraction rule also make an over-approximation (rule 14). Rules 10, 11, 12 reflect the SARE semantic, which view each SARE expression as an array. In particular, $[\![0]\!]$ is viewed as an array filled with zeros with the same dimension as the assigned array, hence its region of interest is empty. Finally, for *reductions* $A[i] = \sum_{i=\Pi(j), j\in D} Expr(j)$, we keep only the indices i of A such that at least one $Expr(j) \neq 0$. This is exactly the set of indices:

$$\{i \mid \exists j \in D : \ i = \Pi(j), Expr(j) \neq 0\} = \Pi([\![Expr]\!] \cap D)$$

Equivalently expressed as $[\![Expr]\!].\{j \to \Pi(j) \mid j \in D\}$. Note that the set of reduction iterations would exactly be $[\![Expr]\!]$.

Example (cont'd). On example 1, we obtain the equation, whose evaluation gives directly the result:

$$\begin{aligned}[\![C]\!] = ([\![A]\!].\{(i,k) \to (i,j,k) \mid \Delta_2\} \cap \\ [\![B]\!].\{(k,j) \to (i,j,k) \mid \Delta_2\}) \\ .\{(i,j,k) \to (i,j) \mid \Delta_2\}\end{aligned}$$

Where Δ_2 is a shortcut for $1 \leq i,j,k \leq N$. On the remainder of this paper, we will focus on example 2 whose recurrence requires additional steps. After simplification, the compilation rules lead to the following sparsity equations:

$$\begin{aligned}[\![S]\!] &= [\![In]\!].\{(i) \to (i) \mid 1 \leq i \leq N-1\} \\ [\![T]\!] &= [\![T]\!].\{(t-1,i-1) \to (t,i) \mid t,i \geq 2 \wedge \Delta_1\} \cup \\ &\quad [\![T]\!].\{(t-1,i) \to (t,i) \mid \neg(t=1) \wedge \Delta_1\} \cup \\ &\quad [\![S]\!].\{(i-1) \to (t,i) \mid (t=1 \text{ or } i=1) \wedge \Delta_1\} \cup \\ &\quad [\![S]\!].\{(i) \to (t,i) \mid t=1 \wedge \Delta_1\}\end{aligned}$$

Where Δ_1 is a shortcut for $(t,i) \in \mathcal{D}_T$: $1 \leq t \leq M, 1 \leq i \leq N-1$. Intuitively, the two terms with $[\![S]\!].\{\ldots\}$ *initialize* the non-zero iterations for $t = 1$, then the two terms with $[\![T]\!].\{\ldots\}$ *propagate* the non-zero iterations across dataflow dependences $\langle T, t-1, i-1\rangle \to \langle T, t, i\rangle$ and $\langle T, t-1, i\rangle \to \langle T, t, i\rangle$ encoded as affine relations.

Resolution. In general, we obtain a *fixpoint* equation, since the solution $X = ([\![S]\!], [\![T]\!])$ is defined as some function of itself: $X = F(X)$. When F is monotonic and the input domains (here $[\![In]\!]$) are finite (non-parametrized), the smallest

$$[\![Array]\!] = [\![Expr]\!] \text{ if } Array := i \mapsto Expr \tag{9}$$

$$[\![Constant]\!] = \begin{cases} \{i \mid \texttt{true}\} \text{ if } Constant \neq 0 \\ \{i \mid \texttt{false}\} \text{ if } Constant = 0 \end{cases} \tag{10}$$

$$[\![Array[\phi(i)]]\!] = [\![Array]\!].\{\phi(i) \longrightarrow i \mid \texttt{true}\} \tag{11}$$

$$[\![\texttt{if } Cond \texttt{ then } Expr_1 \texttt{ else } Expr_2]\!] = [\![Expr_1]\!].\{i \longrightarrow i \mid Cond\} \cup [\![Expr_2]\!].\{i \longrightarrow i \mid \neg Cond\} \tag{12}$$

$$[\![Expr_1 + Expr_2]\!] = [\![Expr_1]\!] \cup [\![Expr_2]\!] \tag{13}$$

$$[\![Expr_1 - Expr_2]\!] = [\![Expr_1]\!] \cup [\![Expr_2]\!] \tag{14}$$

$$[\![Expr_1 * Expr_2]\!] = [\![Expr_1]\!] \cap [\![Expr_2]\!] \tag{15}$$

$$[\![\bot]\!] = [\![0]\!] \tag{16}$$

$$[\![\sum_{i=\Pi(j), j\in D} Expr]\!] = [\![Expr]\!].\{j \longrightarrow \Pi(j) \mid j \in D\} \tag{17}$$

Figure 6. Compilation rules to derive sparsity equations

solution might be computed thanks to Kleene iterations [29]:

$$X = \bigsqcup_{n\in\mathbb{N}} F^n(\bot) \tag{18}$$

where $\bot = (\emptyset, \emptyset)$ and the operator $\sqcup$ is the componentwise union: $([\![S]\!], [\![T]\!]) \sqcup ([\![S']\!], [\![T']\!]) = ([\![S]\!] \cup [\![S']\!], [\![T]\!] \cup [\![T']\!])$. However, this would amount to execute the program which is not interesting as the non-zero inputs are usually very large. Instead, we propose to resolve the sparsity equations by finding a *computable closed form* for $[\![S]\!]$ and $[\![T]\!]$. To do so, we rephrase them as *language equations*.

5.2 Abstraction to Language Equations

We consider each relation as a *letter* and each indice set $[\![S]\!]$ as a *language* L_S:

$$\begin{aligned} L_S &= L_{In}.a \\ L_T &= L_T.(b \cup c) \cup L_S.(d \cup e) \end{aligned}$$

where $a := \{(i) \longrightarrow (i) \mid 1 \leq i \leq N-1\}$, $b := \{(t-1, i-1) \longrightarrow (t, i) \mid t, i \geq 2 \wedge \Delta_1\}$, $c := \{(t-1, i) \longrightarrow (t, i) \mid \neg(t = 1) \wedge \Delta_1\}$, $d := \{(i-1) \longrightarrow (t, i) \mid (t = 1 \text{ or } i = 1) \wedge \Delta_1\}$ and $e := \{(i) \longrightarrow (t, i) \mid t = 1 \wedge \Delta_1\}$. This way, each word $u = \ell_1 \ldots \ell_n$ represents the relation $R_1.\ldots.R_n$ where ℓ_i is the letter *abstracting* the relation R_i.

Provided the intersections are hidden in a letter, we obtain *language equations*, which might be solved directly, using the classical resolution method with Arden's lemma from formal language theory [23]:

$$\begin{aligned} L_S &= L_{In}.a \\ L_T &= L_S.(d \cup e).(b \cup c)^* \end{aligned}$$

Formally, our abstraction is defined as a mapping α from affine relations to regular expressions with the following inductive rules:

$$\alpha(R_1 \cup R_2) = \alpha(R_1) \cup \alpha(R_2) \tag{19}$$

$$\alpha(R_1 \cap R_2) = \ell_{R_1 \cap R_2} \tag{20}$$

$$\alpha(R.u) = \alpha(R).\ell_u \tag{21}$$

$$\alpha([\![S]\!]) = L_S \tag{22}$$

Rule 19 abstracts the union of affine relation as a language union. Rule 20 abstracts away an intersection as a letter $\ell_{R_1 \cap R_2}$. Since abstraction is applied recursively on a relation expression, the application $\alpha(R_1 \cap (R_2 \cup (R_3 \cap R_4)))$ would abstract away the whole expression $R_1 \cap (R_2 \cup (R_3 \cap R_4))$ as a letter. Rule 21 abstracts away the relation u as a letter ℓ_u and the composition as a concatenation. This way, a word $\ell_{u_1} \ldots \ell_{u_n}$ abstracts the relation composition $u_1.\ldots.u_n$ as explained above. Rule 22 translate the affine relation *name* $[\![S]\!]$ to the language *name* L_S. Finally, each equation $[\![S]\!] = R$ is abstracted as $L_S = \alpha(R)$.

5.3 Concretization to Sparsity Equations

Once the language equations are resolved, we get back to the affine world by substituting back letters ℓ_R to affine relations R, language names L_S to relation names $[\![S]\!]$ and by rephrasing the operators $\cup$, . and $*$. This is achieved by a *concretization* operator γ which maps regular expressions to affine relations by using the following rules:

$$\gamma(\ell_R) = R \tag{23}$$

$$\gamma(L_1.L_2) = \gamma(L_1).\gamma(L_2) \tag{24}$$

$$\gamma(L_1 \cup L_2) = \gamma(L_1) \cup \gamma(L_2) \tag{25}$$

$$\gamma(L^*) = \gamma(L)^* \tag{26}$$

$$\gamma(L_S) = [\![S]\!] \tag{27}$$

We substitute each letter ℓ_R by the affine relation R (Rule 23), each language name L_S by its counterpart $[\![S]\!]$ (Rule 27). Then we translate back the language union as a relation union (Rule 25) and the concatenation as a relation composition

(Rule 24). The interpretation of the Kleene star L^* is then:

$$\begin{aligned}\gamma(L^*) &= \gamma(\bigcup_{n\in\mathbb{N}} L^n) \quad (* \text{ definition})\\ &= \bigcup_{n\in\mathbb{N}} \gamma(L^n) \quad (\text{Rule 25})\\ &= \bigcup_{n\in\mathbb{N}} \gamma(L)^n \quad (\text{Rule 24})\\ &= \gamma(L)^*\end{aligned}$$

Hence, the Kleene star L^* translates back to the *transitive closure* of the affine relation $\gamma(L)$ (Rule 26). Although transitive closures of affine relations are not computable in general, the dependence patterns are usually simple enough to conclude (e.g. $(i,j,k-1) \rightarrow (i,j,k)$). *Polyhedral calculators* such as omega or isl provide efficient heuristics, which will be used for calculating these relations. This is the *key part* of our approach.

Example (cont'd). Applying the rules, we get the following concretization:

$$\begin{aligned}[\![S]\!] &= [\![In]\!].\{(i) \rightarrow (i) \mid 1 \le i \le N-1\}\\ [\![T]\!] &= [\![S]\!].\Big(\{(i-1) \rightarrow (t,i) \mid (t=1 \text{ or } i=1) \wedge \Delta_1\} \cup\\ &\qquad \{(i) \rightarrow (t,i) \mid t=1 \wedge \Delta_1\}\Big) .\\ &\quad \Big(\{(t-1,i-1) \rightarrow (t,i) \mid t,i \ge 2 \wedge \Delta_1\} \cup\\ &\qquad \{(t-1,i) \rightarrow (t,i) \mid \neg(t=1) \wedge \Delta_1\}\Big)^*\end{aligned}$$

5.4 Evaluating Sparsity Equations

After concretization, we obtained *simplified* sparsity equations: closed forms have been computed – as much as possible – for domains $[\![S]\!]$. When there is *no dependence cycle* between equations, we can compute an evaluation order and compute incrementally each domain $[\![S]\!]$ using a polyhedral calculator [24, 43].

Example (cont'd). There is a dependence $[\![S]\!] \rightarrow [\![T]\!]$, since $[\![T]\!]$ uses $[\![S]\!]$. Hence, we evaluate $[\![S]\!]$ first. Following the example depicted on Figure 2.(b), we take $N=5$, $M=4$ and we assume In to have a single non-zero indice 3: $[\![In]\!] = \{() \rightarrow (i) \mid i=3\}$. The computation of the relation gives $[\![S]\!] = \{() \rightarrow (i) \mid i=3\}$. Then, we can compute $[\![T]\!] = \{() \rightarrow (t,i) \mid t \ge 1, 3 \le i \le 4\}$. In particular, the transitive closure of $\gamma(b \cup c)$ can be computed exactly and gives $\gamma(b \cup c)^* = \{(t,i) \rightarrow (t',i') \mid t > t', i'-t' \le i-t, (t,i),(t',i') \in \mathcal{D}_T\}$, which means that (t',i') belongs to the *cone* generated by dependence vectors $(1,0)$ (read `A[t-1,i]`) and $(1,1)$ (read `A[t-1,i-1]`) starting from origin point (t,i).

Scalability. We used a polyhedral calculator to validate the soundness of our approach. However, polyhedral calculators are not meant to process integer sets with millions of elements. If the simplified sparsity equations are $([\![S_1]\!],\ldots,[\![S_n]\!]) = \Phi([\![I_1]\!],\ldots,[\![I_m]\!])$, the polyhedral calculator should *only* be used to simplify Φ as much as possible. Then, the efficient evaluation of the simplified Φ on inputs $[\![I_1]\!],\ldots,[\![I_m]\!]$ could take profit of the *massive data parallelism*. For instance, the composition $(\cup_i R_i).u$ might distribute: $\cup_i R_i.u$. Then, each $R_i.u$ might be evaluated in parallel. This part is out of the scope of this paper and is left for future work.

Cycles handling. When dependence cycles remains, this means that a closed form could not be computed for some domains $[\![S]\!]$. We obtained only a *partial* evaluation of domains. To complete the evaluation, *simplified* equations will have to be executed. A direct method would be to execute the Kleene iterations (Eq. 18) on the *simplified* equations. A more sophisticated method would be to generate an efficient evaluator using polyhedral code generation. This part will also be addressed in a future work.

6 Experimental Evaluation

This section presents the experimental evaluation of our propagation algorithm Section. 6.1 presents the experimental setup. Then, Section 6.2 presents an assessment of the scalability and the accuracy of our method.

6.1 Setup

We have applied our method on the following kernels. The kernels *jacobi-1d*, *jacobi-2d*, *syrk* and *syr2k* are from the Polybench/C suite [31]:

- **Sparse Matrix Matrix multiplication (SpMM)** computes $C = A \times B$ from a *sparse* matrix A and a *dense* matrix B.
- **SpGEMM** computes $C := \beta \times C + \alpha A \times B$ from two sparse matrices A, B and two scalars α and β, as the original BLAS kernel.
- **Jacobi 1D and 2D** are order 1 stencils resp. on 1-D and 2-D array.
- **Syrk and Syr2k** are BLAS kernels performing a symmetric rank-k update.

We did not implement the reduction handling so far, hence all these examples are expressed with recurrences. We have evaluated our approach on sparse matrices filled with random regions of interest. From a matrix A, we produce r square regions $R(x_k,y_k,t_k) = \{(i,j) \mid x_k \le i \le x_k+t_k \wedge y_k \le j \le y_k+t_k\}$ where x_k, y_k are the coordinates and t_k the block size. These constraints are then gathered to obtain the final input matrix description $[\![M]\!] = \bigcup_{k=1}^{r} R(x_k,y_k,t_k)$. The characteristics of the matrices used in our experiments are summarized on Table 1.

The final sets are computed with the `iscc` tool [44], using a computer equipped with an Intel(R) Core(TM) i7-10750H CPU and 16 GB RAM DDR4.

n	r
3000	100
15000	500
30000	1000
60000	2000
90000	3000
120000	4000
150000	5000
300000	10000
600000	20000

Table 1. Sparse matrices sizes (n) and number of regions (r). Regions can contain several non-zero points

6.2 Results

This section assesses the scalability and the accuracy of our approach over randomly generated sparse matrices.

Scalability. Figure 7 depicts the execution time on our randomly generated matrices. The horizontal axis gives the number of regions r while the vertical axis gives the timings in second. All tests taking more than one hour have been aborted. Our solution is very efficient for a small number of regions. But not surprisingly from the complexity of the ISL solver, the execution time increases with the number of regions. The difference in execution time between the examples might be explained by the number of union and intersections described in the sparsity equations. Overall, our method scales pretty well regarding the matrix size considered, but perform badly when the number of dependences grow. Indeed for such cases, we can expect the overhead of our analysis to be largely dominated by the computation time. One surprising and very promising result we found is for SpMM: scalabity was the same, but even the biggest matrix only took 14 seconds to compute.

These results showed that the bottleneck of our method might be brought by the polyhedral calculator, which spends a lot of time resolving unions and intersections.

Accuracy. The computation of transitive closure of affine relations might cause further inaccuracy, as the heuristic used may possibly over-approximate the results [9, 44]. We show that this is sufficient for our purpose. Specifically, the *accuracy* is the ratio between the number of *computed* non-null indices $[\![M]\!]$ and the number of *actual* non-null indices $\mathcal{A}[\![M]\!]$:

$$\sum_{\text{Array } M} \text{card}[\![M]\!] / \sum_{\text{Array } M} \text{card}\,\mathcal{A}[\![M]\!]$$

We tested the accuracy by using the cardinal fonctionality of ISCC. We however did not tested SpGEMM, Jacobi-2D, and syr2k-2D: Indeed, the accuracy computation failed – `iscc` did not succeed to compute the cardinals. Instead, we focused on *sparse matrix-multiply*, *jacobi-1d* and *lu*. The results obtained for *sparse-matrix-multiply* are exact. For *jacobi-1d*, the accuracy is not exactly 1 but arround 1.0001. This is due to the overapproximation of transitive closures made by `iscc`. Although the computation the accuracy failed on *gemm*, the accuracy is likely to 1, as the transitive closure required is the same as for *sparse-matrix-multiply*.

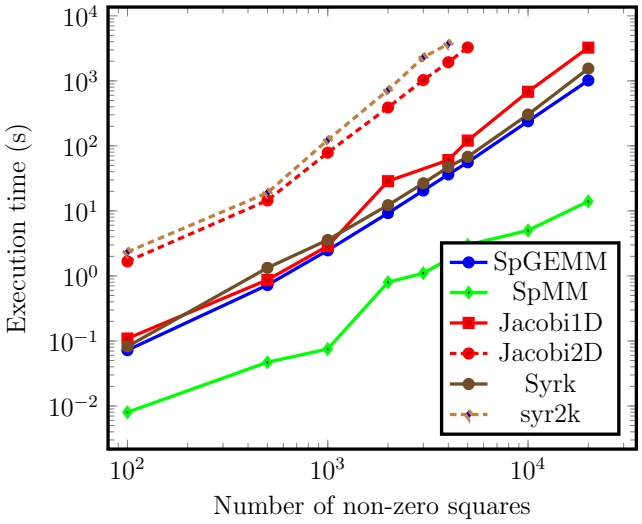


Figure 7. Scalability analysis

7 Conclusion and future work

In this paper, we have presented an algorithm for specializing automatically a dense polyhedral program on sparse data. Our algorithm relies on setting and solving flow equations to propagate the sparsity across the computation in order to identify useful computation. Experimental validation confirms the accuracy of our approach. We believe that propagation of sparse data will be particularly useful in programs with consecutive kernels, that can be found in machine learning or iterative solvers. In the future, we plan to improve the evaluation step by exploiting the large data parallelism. Also, we will address the case where only a partial evaluation was found (cycles). We plan to incorporate our specialization algorithm into a general framework for automatic parallelization of sparse code and generation. We also intend to extend the method to tensor computation, and also investigate solver kernels based on gaussian elimination. Also, inspector/executor methods to split the specialization between compile-time and runtime to further reduce the runtime overhead could be very interesting. Finally, we aim to ease the number of the polyhedron constraints brought with the polyhedral representation by taking inspirations such as sparse block formats [22, 30] to group consecutive values.

Acknowledgments

This work was funded by the PEPR NumPEx through the ExaSoft project.

References

[1] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. [n. d.]. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal QC Canada, 2023-02-17). ACM, 41–54. https://doi.org/10.1145/3579990.3580020

[2] Christophe Alias and Alexandru Plesco. 2021. Data-aware process networks. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 1–11.

[3] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. [n. d.]. Generating Piecewise-Regular Code from Irregular Structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix AZ USA, 2019-06-08). ACM, 625–639. https://doi.org/10.1145/3314221.3314615

[4] Cédric Bastoul. 2003. Efficient Code Generation for Automatic Parallelization and Optimization. In *2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), 13-14 October 2003, Ljubljana, Slovenia*. 23–30.

[5] Nathan Bell and Michael Garland. [n. d.]. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland Oregon, 2009-11-14). ACM, 1–11. https://doi.org/10.1145/1654059.1654078

[6] Aart JC Bik and Harry AG Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*. 416–424.

[7] Aart J. C. Bik and Harry A. G. Wijshoff. 1993. Compilation Techniques for Sparse Matrix Computations. In *Proceedings of the 7th International Conference on Supercomputing* (Tokyo Japan). ACM, 416–424. https://doi.org/10.1145/165939.166023

[8] Aart J. C. Bik and Harry A. G. Wijshoff. 1993. On Automatic Data Structure Selection and Code Generation for Sparse Computations. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 57–75.

[9] Bernard Boigelot. 1998. *Symbolic methods for exploring infinite state spaces.* Ph. D. Dissertation. Université de Liège.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 101–113. https://doi.org/10.1145/1375581.1375595

[11] Kazem Cheshmi, Zachary Cetenic, and Maryam Mehri Dehnavi. 2022. Vectorizing Sparse Matrix Computations with Partially-Strided Codelets. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC'22)*. IEEE Press.

[12] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri 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* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3126908.3126936

[13] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 115–126. https://doi.org/10.1145/1693453.1693471

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. [n. d.]. Format Abstraction for Sparse Tensor Algebra Compilers. In *Proceedings of the ACM on Programming Languages* (2018-10-24), Vol. 2. 1–30. Issue OOPSLA. https://doi.org/10.1145/3276493

[15] Charles Consel and Olivier Danvy. 1993. Tutorial Notes on Partial Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 493–501. https://doi.org/10.1145/158511.158707

[16] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[17] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. https://doi.org/10.1007/BF01407835

[18] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420. https://doi.org/10.1007/BF01379404

[19] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*. 1581–1592.

[20] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[21] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. 2023. Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 160–171. https://doi.org/10.1145/3559009.3569668

[22] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *Int. J. High Perform. Comput. Appl.* 18 (02 2004), 135–158. https://doi.org/10.1177/1094342004041296

[23] Jeffrey D. Ullman John E. Hopcroft, Rajeev Motwani. [n. d.]. *Introduction to automata theory, languages, and computation.* Addison-Wesley.

[24] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.

[25] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. [n. d.]. The Tensor Algebra Compiler. In *Proceedings of the ACM on Programming Languages* (2017-10-12), Vol. 1. 1–29. Issue OOPSLA. https://doi.org/10.1145/3133901

[26] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[27] Daniel Langr and Pavel Tvrdík. 2016. Evaluation Criteria for Sparse Matrix Storage Formats. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27. 428–440.

[28] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. 2017. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4192.

[29] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis.* springer.

[30] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. [n. d.]. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul Republic of Korea, 2022-04-02). ACM, 90–106.

[31] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]* (2012).

[32] William Pugh and Tatiana Shpeisman. [n. d.]. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *Languages and Compilers for Parallel Computing*, Siddhartha Chatterjee, Jan F. Prins, Larry Carter, Jeanne Ferrante, Zhiyuan Li, David Sehr, and Pen-Chung Yew (Eds.). Vol. 1656. Springer Berlin Heidelberg, 213–229.

[33] Patrice Quinton and Vincent van Dongen. 1989. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology* 1, 2 (1989), 95–113.

[34] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. 1986. On synthesizing systolic arrays from Recurrence Equations with Linear Dependencies. In *Foundations of Software Technology and Theoretical Computer Science*, Kesav V. Nori (Ed.). Lecture Notes in Computer Science, Vol. 241. Springer Berlin Heidelberg, 488–503.

[35] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. [n. d.]. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa Israel, 2016-09-11). ACM, 247–259. https://doi.org/10.1145/2967938.2967943

[36] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. [n. d.]. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach California USA, 2015-06-08). ACM, 99–108. https://doi.org/10.1145/2751205.2751244

[37] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-time composition of run-time data and iteration reorderings. *ACM SIGPLAN Notices* 38, 5 (2003), 91–102.

[38] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. [n. d.]. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 ([n. d.]), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[39] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 99 (2018), 1–15.

[40] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the sparse polyhedral framework. *Parallel Comput.* 53 (2016), 32–57.

[41] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. [n. d.]. *A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR.* arXiv:2102.05187 [cs] http://arxiv.org/abs/2102.05187

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

[43] Sven Verdoolaege. 2010. ISL: An Integer Set Library for the Polyhedral Model. In *ICMS*, Vol. 6327. Springer, 299–302.

[44] Sven Verdoolaege. 2011. Counting affine calculator and applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Charmonix, France.*

[45] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *Proc. ACM Program. Lang.* 7, PLDI, Article 188 (jun 2023), 26 pages. https://doi.org/10.1145/3591302

[46] Jie Zhao, Michael Kruse, and Albert Cohen. 2018. A Polyhedral Compilation Framework for Loops with Dynamic Data-Dependent Bounds. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3178372.3179509

[47] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054