# Partial Evaluation of Dense Code on Sparse Structures

Gabriel Dehame, Christophe Alias, Alec Sadler

# Partial Evaluation of Dense Code on Sparse Structures

**Gabriel Dehame, Christophe Alias, Alec Sadler**

**RESEARCH REPORT**
**N° 9534**
December 2023
Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9534--FR+ENG

# Partial Evaluation of Dense Code on Sparse Structures

Gabriel Dehame*, Christophe Alias†, Alec Sadler‡

Project-Team Cash

Research Report n° 9534 — December 2023 — 16 pages

**Abstract:** Most HPC computations process sparse tensors. The resulting code is highly dynamic, which makes code optimization 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. Then, to specialize that code on the sparse input structure. In this paper, we propose a novel approach to apply that specialization. The key ingredient of our algorithm is the transitive closure of affine relation, for which efficient and accurate heuristics exist. Experimental evaluation shows the effectiveness of our approach.

**Key-words:** Code optimization, Sparse code, Specialization

* Inria/ENS-Lyon/UCBL/CNRS, gabriel.dehame@ens-lyon.fr
† Inria/ENS-Lyon/UCBL/CNRS, *corresponding author*, christophe.alias@inria.fr
‡ Inria/ENS-Lyon/UCBL/CNRS, alec.sadler@inria.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

# Evaluation partielle de codes denses sur des structures creuses

**Résumé :** La plupart des calculs HPC manipulent des tenseurs creux. Le code résultant est très dynamique, ce qui rend l'optimisation du code assez difficile. Une méthode est de partir de la spécification dense originale, qui est généralement beaucoup plus réguliére et se prête bien mieux à l'optimisation de code; et ensuite, de spécialiser le code sur la structure d'entrée. Dans cet rapport, nous proposons une nouvelle approche pour appliquer cette spécialisation. L'ingrédient clé de notre algorithme est la fermeture transitive d'une relation affine, pour laquelle des heuristiques efficaces et précises existent. L'évaluation expérimentale confirme l'efficacité de notre approche.

**Mots-clés :** Optimisation de code, calcul creux, spécialisation

# 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 parallelisation focuses on programming automatically parallel computers from a sequential specification. In the past decades, a general unified framework, the *polyhedral model* [20], 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 [3, 25, 12].

However, most kernels of interest in machine learning and high-performance computing manipulate *sparse* tensors. 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: *runtime parallelization* of the sparse code and *compile-time code generation* from a dense code specification assuming sparse data.

*Runtime sparse parallelizers* [33, 34, 31] 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* produce an optimized sparse code from a dense code specification with annotations. Since the seminal work of Bik [10] on efficient sparse data structure selection, several approaches were proposed to produce a sparse code with a minimal amount of computations. Kjolstad [23] proposes a 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 [21]. While these approaches operate on generic sparse data structures, Augustine [5] propose to *specialize* a matrix-vector product on a sparse matrix at compile-time, while applying polyhedral optimizations.

In this paper, we propose a general method for the automatic specialization of a dense code on sparse tensors. Our approach has many applications in sparse code optimization. It could be used at compile time to produce a versionning on different sparse matrix shapes; or at runtime to specialize dynamicaly a code once the sparse structure is known and then enable deeper optimization than those taking the sparse code without any knowledge of the original dense code. Specifically, we make the following contributions:

- We propose an algorithm for specializing dense code from a sparse data description; based on the direct resolution of fix-point equations. Our resolution technique relies on langage theory results.
- We have evaluated our approach on large sparse matrices. The results show that our method is scalable and accurate.

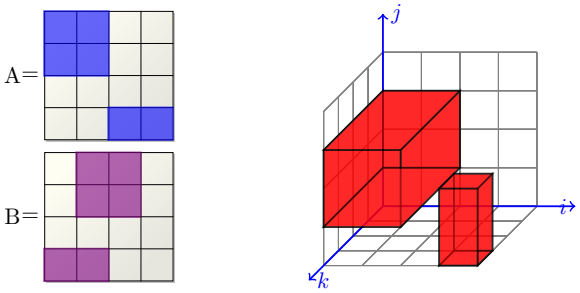
This paper is structured as follows. Section 3 introduces the polyhedral model and the mathematical tools used in the algorithm description. Section 4 presents the related work. Section 5 presents our specialization algorithm. Section 6 presents our experimental validation. Finally, Section 7 concludes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

# 2 Motivating Example

Consider the matrix multiplication kernel depicted on Figure 1.(a). Given input sparse matrices, our goal is to propagate the sparsity across the computation, and keep only the useful computations depicted in (b). Many sparse formats exists including CSR, ELLPACK, or CSC [13, 1]. The selection of the best format depends on scheduling and dependence patterns [9]. We assume

```
    for(i=1; i<=N; i++)
      for(j=1; j<=N; j++) {
R:      C[i][j] = 0;
        for(k=1; k<=N; k++)
S:        C[i][j] += A[i][k]*B[k][j];
      }
```

(a) Dense kernel




(b) Sparse specialization

Figure 1: Motivating example

that sparse matrices are described as a union of convex polyhedra as depicted in (b). How to obtain this union from a classical sparse format is out of the scope of this paper; and has already been investigated [5]. There is clearly not a single solution, many trade-offs must be explored. In particular, the regions can be sparse themselves, hence a density threeshold must be set.

The outcome of our algorithm is the specialized program. On our example, we would only have relevant loop iterations depicted in red on the picture. It is actually sufficient to have a description of the counter ranges as constraints coupled with a schedule description, as code generation techniques are able to produce automatically a loop kernel from that specification [6].

# 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 [28, 27, 17, 18, 19, 20] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [20] and data locality improvement [12]. 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 [17], scheduling [19] or loop tiling [12] 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)** The matrix multiplication is clearly a polyhedral program with two statements $R$ and $S$, whose iteration domains are *polyhedra parametrized by* $N$: $\mathcal{D}_R = \{(i,j) \mid 1 \leq i,j \leq N\}$, $\mathcal{D}_S = \{(i,j,k) \mid 1 \leq i,j,k \leq N\}$.

## 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 [17]. 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)** On our example, the SARE is:

$$\begin{array}{rcl} C[i,j] & = & \texttt{if } 1 \leq i,j \leq N \texttt{ then } S[i,j,N] \\ & & \texttt{else } \bot \\ S[i,j,k] & = & \texttt{if } 1 \leq i,j,k \leq N \texttt{ then} \\ & & (\texttt{if } k = 1 \texttt{ then } 0 \\ & & \texttt{ else } S[i,j,k-1]) + A[i,k] * B[k,j] \\ & & \texttt{else } \bot \end{array}$$

To simplify the presentation, the statement $R$ has been removed by a constant propagation.

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):

$$\begin{array}{rcl} C & := & \gamma(i,j).\texttt{if } 1 \leq i,j \leq N \texttt{ then } \gamma(i,j).S[i,j,N] \\ & & \texttt{else } \bot \\ S & := & \gamma(i,j,k).\texttt{if } 1 \leq i,j,k \leq N \texttt{ then} \\ & & (\texttt{if } k = 1 \texttt{ then } 0 \\ & & \texttt{ else } \gamma(i,j,k).S[i,j,k-1])) + \\ & & \gamma(i,j,k).A[i,k] * \gamma(i,j,k).B[k,j] \\ & & \texttt{else } \bot \end{array}$$

Note the syntactic sugar $\gamma i.e$, which recalls that $e$ *actually denotes the mapping* $i \mapsto e$. In particular, $\gamma i.S[\phi(i)]$ is a notation for $S \circ \phi$ which recalls $i$ in the definition of $\phi$, written directly as the expression $\phi(i)$. This will allow to write convenient syntax-directed translation rules later in the paper. However, this should not be confused with the $\lambda$ notation of the $\lambda$ calculus.

**Syntax** The SARE syntax is depicted on Figure 2, where *Constant* denotes a constant value, *Array* denotes an array identifier, $\phi$ denotes an affine function and *Cond* denotes an affine condition. We assume the SARE to be well formed. The affine expressions defining $\phi$ and *Cond*

$$
\begin{aligned}
\mathit{Sare} &::= \mathit{Def}+ \\
\mathit{Def} &::= \mathit{Array} := \gamma i.\mathit{Expr} \\
\mathit{Expr} &::= \mathit{Constant} \mid \gamma i.\mathit{Array}[\phi(i)] \\
&\quad \mid \texttt{if } \mathit{Cond} \texttt{ then } \mathit{Expr} \texttt{ else } \mathit{Expr} \\
&\quad \mid \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} * \mathit{Expr} \mid \bot
\end{aligned}
$$

Figure 2: SARE syntax

$$
\begin{aligned}
\mathcal{S}[\![\mathit{Array}]\!](x) &= \mathcal{S}[\![\mathit{Expr}]\!](x) \text{ if } \mathit{Array} := \gamma i.\mathit{Expr} \text{ and } x \in \mathbb{Z}^{\dim i} && (1)\\
\mathcal{S}[\![\mathit{Constant}]\!](x) &= \mathcal{E}[\![\mathit{Constant}]\!] && (2)\\
\mathcal{S}[\![\gamma i.\mathit{Array}[\phi(i)]]\!](x) &= \mathcal{S}[\![\mathit{Array}]\!](\phi(x)) && (3)\\
\mathcal{S}[\![\texttt{if } \mathit{Cond} \texttt{ then } \mathit{Expr}_1 \texttt{ else } \mathit{Expr}_2]\!](x) &= \begin{cases} \mathcal{S}[\![\mathit{Expr}_1]\!](x) \text{ if } \mathcal{B}[\![\mathit{Cond}]\!](x) \text{ is true,} \\ \mathcal{S}[\![\mathit{Expr}_2]\!](x) \text{ otherwise} \end{cases} && (4)\\
\mathcal{S}[\![\mathit{Expr}_1 + \mathit{Expr}_2]\!](x) &= \mathcal{S}[\![\mathit{Expr}_1]\!](x) + \mathcal{S}[\![\mathit{Expr}_2]\!](x) && (5)\\
\mathcal{S}[\![\mathit{Expr}_1 * \mathit{Expr}_2]\!](x) &= \mathcal{S}[\![\mathit{Expr}_1]\!](x)\mathcal{S}[\![\mathit{Expr}_2]\!](x) && (6)\\
\mathcal{S}[\![\bot]\!](x) &= \bot && (7)
\end{aligned}
$$

Figure 3: SARE semantics

should only involve the indices of the enclosing $\gamma$ and the structure parameters (e.g. $N$). 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 3. 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 $\gamma i.\mathit{Array}[\phi(i)]$ 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.

## 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 $\Phi$ (arithmetic over $(\mathbb{Z}, +)$) with $n+p$ free variables with $(x_1, \ldots x_n) \rightarrow (y_1, \ldots y_p)$ iff $x, y$ satisfy $\Phi$: $(x_1, \ldots, x_n, y_1, \ldots, y_p) \vdash \Phi$. Usually, those relations are referred to as *affine relations*, as they manipulate affine forms.

**Example (cont'd)** The following affine relation:

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

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

In this paper, we will use extensively affine relations and particularly their transitive closure (to emulate loops) $\to^* = \bigcup_{k \in \mathbb{N}} \to^k$. 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 [11, 37]. 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: $\{() \to x \mid x \in P\}$. For instance, the set of iteration reading $A[0][0]$ could simply be obtained with $\{() \to (0,0)\}.\{(i,k) \to (i,j,k) \mid 1 \leq i,j,k \leq N\}$.

# 4 Related Work

In this section, we first outline the work arround the generation and the optimization of sparse code; then we outline the work on code specialization.

Our work can be considered as an approach for co-iteration over two or more sparse tensors. Efficient co-iteration support and specialization are essential for performances, 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 [9, 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 [26] 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 [22], 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 [15], which can prove to be more efficient on modern GPUs [7, 30, 14].

**Sparse code optimizers** The polyhedral model 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 [4] used trace reconstruction in order to exploit patterns for sparse matrix vector multiplication, and Zhao et al. [38] computes static upper bound and model control dependences of non-affine loop bounds. The Sparse Polyhedral Framework [32] combines the polyhedral model with inspector-executor methods to target non-affine loops. Venkat et al.[36] describe loop transformations to compose sparse layout to polyhedral scanning. Zhao et al. [39] 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 [29] and COMET [35] describe data reordering techniques to improve locality during computation. Looplet [2],

$$[\![Array]\!] = [\![Expr]\!]_i \text{ if } Array := \gamma i.Expr \tag{8}$$

$$[\![Constant]\!]_i = \begin{cases} \{i \mid \texttt{true}\} \text{ if } Constant \neq 0 \\ \{i \mid \texttt{false}\} \text{ if } Constant = 0 \end{cases} \tag{9}$$

$$[\![\gamma i.Array[\phi(i)]]\!]_i = [\![Array]\!].\{\phi(i) \to i \mid \texttt{true}\} \tag{10}$$

$$[\![\texttt{if } Cond \texttt{ then } Expr_1 \texttt{ else } Expr_2]\!]_i = [\![Expr_1]\!].\{i \to i | Cond\} \cup [\![Expr_2]\!].\{i \to i | \neg Cond\} \tag{11}$$

$$[\![Expr_1 + Expr_2]\!]_i = [\![Expr_1]\!]_i \cup [\![Expr_2]\!]_i \tag{12}$$

$$[\![Expr_1 * Expr_2]\!]_i = [\![Expr_1]\!]_i \cap [\![Expr_2]\!]_i \tag{13}$$

$$[\![\bot]\!]_i = [\![0]\!]_i \tag{14}$$

Figure 4: Compilation rules to derive sparsity equations

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.

**Code specialization** Specialization on sparse data structure is closely intertwined with partial evaluation [16], 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 [24] still prove to be efficient in essential computations, which then need multiple implementations.

# 5 Our Approach

Our method consists of two steps. First, we compute a set of equations whose solution is exactly the regions of interest for each tensor used by the program (Section 5.1). Then, we propose a resolution method (Section 5.2). Sometimes, the equations need to be relaxed to make the resolution possible (Section 5.3).

## 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 4. 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 13), and assuming that $x + y \neq 0$ whenever $x \neq 0$ or $y \neq 0$ (rule 12), which is, of course, an overapproximation as $x + (-x) = 0$ for $x \neq 0$. Rules 9, 10, 11 reflect the SARE semantic, which view each SARE expression as an array. In particular, $[\![0]\!]$ is viewed as an array filled with zeros, hence its region of interest is empty.

**Example (cont'd)** After simplification, the compilation rules lead to the following sparsity equations:

$$\begin{aligned} [\![C]\!] &= [\![S]\!].\{(i,j,N) \to (i,j) \mid \Delta_1\} \\ [\![S]\!] &= [\![S]\!].\{(i,j,k-1) \to (i,j,k) \mid \neg(k=1) \wedge \Delta_2\} \cup \\ &\quad ([\![A]\!].\{(i,k) \to (i,j,k) \mid \Delta_2\} \cap \\ &\quad [\![B]\!].\{(k,j) \to (i,j,k) \mid \Delta_2\}) \end{aligned}$$

Where $\Delta_1$ is a shortcut for $1 \leq i, j \leq N$, and $\Delta_2$ is a shortcut for $1 \leq i, j, k \leq N$. Intuitevely, the intersection $[\![A]\!].\{(i,k) \to (i,j,k) \mid \Delta_2\} \cap [\![B]\!].\{(k,j) \to (i,j,k) \mid \Delta_2\}$ is the set of loop iterations $(i,j,k)$ where the product $A[i][k] * B[k][j]$ is non null. The iteration term $[\![S]\!].\{(i,j,k-1) \to (i,j,k) \mid \neg(k=1) \wedge \Delta_2\}$ propagates the non null iterations forward in the direction of $k$, until the last iteration $k = N$. The results are then collected thanks to $\{(i,j,N) \to (i,j) \mid \Delta_1\}$ as the non null domain of $C$.

## 5.2 Resolving the sparsity equations

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

$$\begin{array}{rcl} L_C & = & L_S.a \\ L_S & = & L_S.b \cup c \end{array}$$

where $a := \{(i,j,N) \to (i,j) \mid \Delta_1\}$, $b := \{(i,j,k-1) \to (i,j,k) \mid \neg(k=1) \wedge \Delta_2\}$ and $c := [\![A]\!].\{(i,k) \to (i,j,k) \mid \Delta_2\} \cap [\![B]\!].\{(k,j) \to (i,j,k) \mid \Delta_2\}$.

Whenever the intersections might be hidden in a letter, we obtain *langage equations*, which might be solved directly, using the classical resolution method from rational language theory:

$$\begin{array}{rcl} L_C & = & L_S.a \\ L_S & = & c.b^* \end{array}$$

Although transitive closures of affine relation are not computable in general, the dependence patterns (e.g. $(i,j,k-1,N) \to (i,j,k,N)$) are usually simple enough to conclude.

**Example (cont'd)** Taking $[\![A]\!] = \{(i,j) \mid 1 \leq i,j \leq 2 \vee (i = 3 \wedge 3 \leq j \leq 4)\}$ and $[\![B]\!] = \{(i,j) \mid (1 \leq i, j-1 \leq 2) \vee (i = 4 \vee 1 \leq j \leq 2)\}$, the computation of the relation gives $[\![S]\!] = \{(i,j,k) \mid (1 \leq i \leq 2 \wedge 2 \leq j \leq 3 \wedge 1 \leq k \leq 4) \vee i = k = 4 \wedge 1 \leq j \leq 2\}$ and $[\![C]\!] = \{(i,j) \mid i = 4 \wedge 1 \leq j \leq 2\}$. In particular, the transitive closure of b can be computed exactly and gives $b^* = \{(i,j,k) \to (i,j,k') \mid 1 \leq i,j,k,k' \leq N \wedge k \leq k'\}$. It contains $\{(i,j,k) \to (i,j,N) \mid \Delta_2\}$ which, combined with the equation defining $C$, leads to the expected result. To obtain the final code, it then suffice to feed a polyhedral code generator with these domains togheter with a valid schedule. For instance, $\theta_S(i,j,k) = (i,j,k)$ and $\theta_C(i,j) = (i,j,N+1)$.

## 5.3 Dealing with intersections

This method no longer works when the intersection occurs *in a recursion*. e.g. $L_S = L_S.b \cap c$. In that case, we simply propose to over-approximate the intersection by bounding the solution: $L_S \subseteq L_S.b$, $L_S \subseteq c$. By iterating this process on the equations, we end-up with *language inequations*, which can be solved by applying an extension of Arden's lemma:

**Lemma 5.1 (Generalized Arden's lemma)** *Consider a rational language $L$ over $\Sigma$ and $u, v \in \Sigma^*$ such that: $L \subseteq L.u$ and $L \subseteq v$.*
*Then, $L \subseteq v.u^*$*

Hence the same resolution process might be applied.

# 6 Evaluation

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

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

Table 1: Matrices used for the experiments. $n \times n$: size, $r$: regions of interest.

## 6.1 Experimental setup

We have applied our method to the following kernels.

- **Matrix multiplication** (MM): computes $C = A \times B$ from two matrices $A$ and $B$.
- **GEMM** (GEMM) computes $C := \beta \times C + \alpha A \times B$ from three matrices $A$, $B$ et $C$ and two scalars $\alpha$ and $\beta$, as the original BLAS kernel.
- **LU decomposition** (LU): computes the LU decomposition kernel of a matrix $A$. The result is directly written on the matrix $A$.
- **Jacobi 1D** (J1D): order 1 stencil on a 1-D array.
- **Jacobi 2D** (J2D): order 1 stencil on a 2-D array.
- **Heat 3D** (Heat): order 1 stencil on a 3-D array.

We have evaluated our approach on sparse matrices filled with random regions of interest. Specifically, from the matrix size $n \times n$ and the desired number of regions of interest $r$, we produce $r$ square regions $R(x_k, y_k, t_k) = \{(i, j) \mid x_k \leq i \leq x_k + t_k \wedge y_k \leq j \leq y_k + t_k\}$ where $x_k$, $y_k$ and $t_k$ are valid random integers and $t_k \in [\![3, n/r]\!]$. These constraints are then gathered to obtain the final input matrix description $[\![M]\!] = \bigcup_{k=1}^{r} R(x_k, y_k, t_k)$ to be passed to our algorithm. The characteristics of the matrices used in our experiments are summarized on Table 1.

The final sets are computed using the `iscc` tool [37] using a computer equipped with an Intel Core i5-10300H Comet Lake and 16 GB RAM DDR4.

## 6.2 Experimental results

We now analyse the scalability of our method on the matrices presented so far; then we analyse the accuracy of our relaxation method.

**Scalability** Figure 5 depicts the execution time on randomly generated matrices. The horizontal axis gives the number of regions $r$ – the corresponding matrix size is provided on Table 1 – while the vertical gives the timing in second. We have decided to abort all tests taking more than one hour.

Our solution is very efficient for a small number of regions, except for *heat-3d*. Not surprisingly, the execution time increases with the number of regions. Especially for *jacobi-1d*,

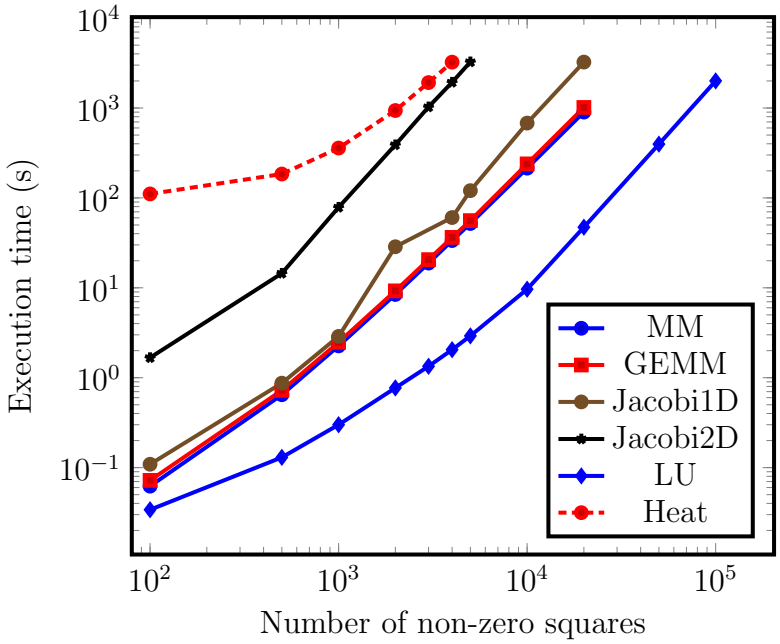



Figure 5: Scalability analysis

*jacobi-2d* and *heat-3d* whose dependence function leads to more complex affine relations. Overall, our method scales pretty well regarding the matrix size considered. Indeed for such cases, we can expect the overhead of our analysis to be largely dominated by the computation time.

Also, if $\mathcal{A}[\![M]\!]$ denotes the expression obtained for the region of interest of $M$ after the possible intersection approximation, the final solution might be written as

$$(\mathcal{A}[\![S_1]\!], \ldots, \mathcal{A}[\![S_n]\!]) = \Phi([\![I_1]\!], \ldots, [\![I_m]\!])$$

where $I_k$ denotes the input arrays and $S_k$ denotes the remaining SARE arrays. The specialization process might be sped-up by precomputing $\Phi$ at compile-time, and leaving only the final application to $[\![I_1]\!], \ldots, [\![I_m]\!]$ to the runtime. How to put this optimization to work is left to future work.

**Accuracy Assessment** We now measure the impact of intersection over-approximation. Note that the computation of transitive closure of affine relations might cause further inaccuracy, as the heuristic used may possibly over-approximate the results [11, 37]. Still, this is sufficient for our purpose. Specifically, the *accuracy* is the ratio between the number of *computed* non-null indices and the number of *actual* non-null indices: $\sum_{\text{Array } M} \text{card}\, \mathcal{A}[\![M]\!] / \sum_{\text{Array } M} \text{card}[\![M]\!]$. Figure 6 depicts the results. We did not test *heat-3d*, *jacobi-2d* and *gemm*. Indeed, the accuracy computation failed – `iscc` did not succeed to compute the cardinals. Instead, we focused on *matrix-multiply*, *jacobi-1d* and *lu*. The results obtained for *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`. For *lu*, an intersection approximation was required, causing an inaccuracy. However, the region computed is still smaller than the original dense matrix (17% of the dense size). In addition, although the computation the accuracy failed on *gemm*, the accuracy is likely to 1, as the transitive closure required is the same as for *matrix-multiply*.

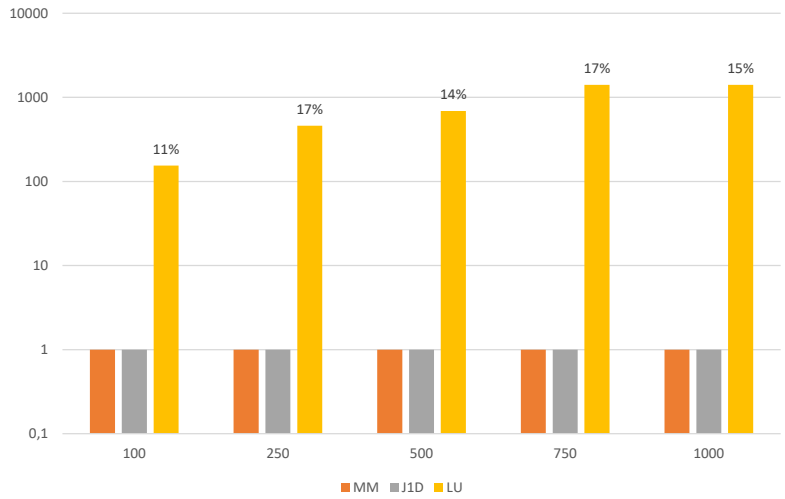



Figure 6: Accuracy assessment, the percents above the bars indicate the ratio computed set / dense set

# 7 Conclusion

In this paper, we have presented an algorithm for specializing automatically a dense code on sparse data. Our algorithm relies on setting and solving flow equations to propagate the sparsity across the computation. The general resolution is undecidable, however, we propose a sound and accurate approximation. Experimental validation confirms the scalability and the accuracy of our approach. In the future, we plan to incorporate our specialization algorithm into a general framework for automatic parallelization of sparse code. We will also investigate methods to split the specialization between compile-time and runtime to further reduce the runtime overhead.

# References

[1] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. An asymptotic cost model for autoscheduling sparse tensor programs, 2021.

[2] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 41–54. ACM.

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

[4] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 625–639. ACM.

[5] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM*

*SIGPLAN Conference on Programming Language Design and Implementation*, pages 625–639, 2019.

[6] Cédric Bastoul. 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*, pages 23–30, 2003.

[7] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM.

[8] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th International Conference on Supercomputing*, pages 416–424. ACM, 1993.

[9] Aart J. C. Bik and Harry A. G. Wijshoff. 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*, pages 57–75, Berlin, Heidelberg, 1993. Springer-Verlag.

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

[11] Bernard Boigelot. *Symbolic methods for exploring infinite state spaces.* PhD thesis, Université de Liège, Liège, Belgium, 1998.

[12] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 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*, pages 101–113, 2008.

[13] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *ACM sigplan notices*, 45(5):115–126, 2010.

[14] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 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*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. Association for Computing Machinery.

[15] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. In *Proceedings of the ACM on Programming Languages*, volume 2, pages 1–30.

[16] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 493–501, New York, NY, USA, 1993. Association for Computing Machinery.

[17] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[18] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[19] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[20] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

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

[22] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. In *Proceedings of the ACM on Programming Languages*, volume 1, pages 1–29.

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

[24] Daniel Langr and Pavel TvrdÃk. Evaluation criteria for sparse matrix storage formats. In *IEEE Transactions on Parallel and Distributed Systems*, volume 27, pages 428–440, 2016.

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

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

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

[28] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 241 of *Lecture Notes in Computer Science*, pages 488–503. Springer Berlin Heidelberg, 1986.

[29] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 247–259. ACM.

[30] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 99–108. ACM.

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

[32] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proceedings of the IEEE*, 106(11):1921–1934.

[33] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, (99):1–15, 2018.

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

[35] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR.

[36] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 521–532, New York, NY, USA, 2015. Association for Computing Machinery.

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

[38] Jie Zhao, Michael Kruse, and Albert Cohen. A polyhedral compilation framework for loops with dynamic data-dependent bounds. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 14–24, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. Polyhedral specification and code generation of sparse tensor contraction with co-iteration. *ACM Trans. Archit. Code Optim.*, 20(1), dec 2022.

# Table des matières

**RESEARCH CENTRE**
**GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399