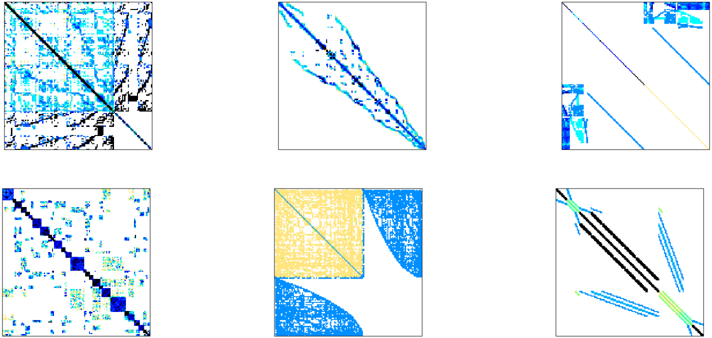
# Automatic Specialization of Polyhedral Programs on Sparse Structure

**Alec Sadler**, Christophe Alias

IMPACT'25, *Barcelona*

# So what are sparse codes ?

- Codes manipulating sparse matrices
- present in pruned ML models, and scientific computing.
- Sparse matrices are too large → store only non-zeros!



Credits: SuiteSparse Collection

- Codes manipulating sparse matrices
- present in pruned ML models, and scientific computing.
- Sparse matrices are too large $\rightarrow$ store only non-zeros!

**Example:**

explicit matrix

$$\begin{bmatrix} x & & & \\ y & & & \\ & & & \\ & & z & t \end{bmatrix}$$

$M$

CSR format

| | | | | |
|---|---|---|---|---|
| *ind* | 0 | 1 | 2 | 4 |

| | | | | |
|---|---|---|---|---|
| *col* | 0 | 0 | 2 | 3 |
| data | $x$ | $y$ | $z$ | $t$ |

$\forall\ ind[i] \leqslant j < ind[i+1]$,
$M_{i,col[j]} = data[j]$

# So what are sparse codes ?

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
  for (k=0; k<P; k++)
   C[i,j]+=A[i,k]*B[k,j];
```

```
for (int i = 0; i < A1_dimension; i↩
    ++) {
  int kA = A2_pos[i];
  int pA2_end = A2_pos[(i + 1)];
  int kB = B1_pos[0];
  int pB1_end = B1_pos[1];

  while (kA < pA2_end && kB < ↩
      pB1_end) {
    int kA0 = A2_crd[kA];
    int kB0 = B1_crd[kB];
    int k = min(kA0,kB0);
    int B1_segend = kB;
    while (B1_segend < pB1_end && ↩
        B1_crd[B1_segend] == k) {
      B1_segend++;
    }
    if (kA0 == k && kB0 == k) {
      for (int jB = kB; jB < ↩
          B1_segend; jB++) {
        int j = B2_crd[jB];
        int jC = i * C2_dimension ↩
            + j;
        C_vals[jC] = C_vals[jC] + ↩
            A_vals[kA] * B_vals[↩
            jB];
      }
    }
    kA += (int)(kA0 == k);
    kB = B1_segend;
  }
}
```

# Challenges

**Dynamic code:** indirections and irregular data accesses create unknown loop trip counts and load balancing.

**Dynamic data:**

- Sparse structures aren't always known in advance.
- They can also change during execution

How might we apply traditional loop transformation to effectively optimize sparse codes?

**Sparse code generation:**

- TACO[Kjolstad,17], formalism used in MLIR[Bik,22].

**Sparse code optimization :**

- Sparse Polyhedral Model [Strout,18]
- runtime data reordering (Sparso[Rong,16], COMET[Tian,21])

**Sparse code specialization :**

- Symbolic analysis (Sympiler[Cheshmi,17], Parsy[Cheshmi,18])
- Specialization in TACO with Looplets[Ahrens,23]
- Piecewise auto-vectorisation[Augustine,19], [Pouchet,23]

# State of the art

**Sparse code generation:**

- TACO[Kjolstad,17], formalism used in MLIR[Bik,22].

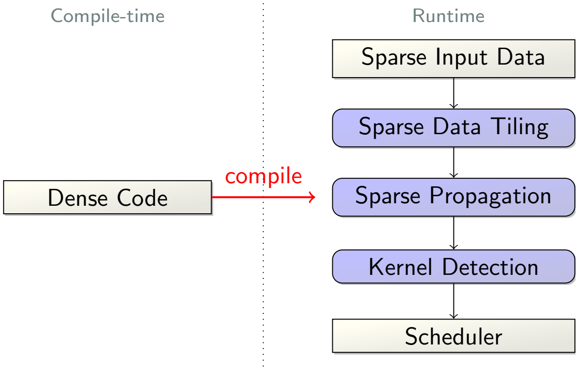
**Sparse code optimization :**

- Sparse Polyhedral Model [Strout,18]
- runtime data reordering (Sparso[Rong,16], COMET[Tian,21])

**Sparse specialization :** ← **Our work!**

- Symbolic analysis (Sympiler[Cheshmi,17], Parsy[Cheshmi,18])
- Specialization in TACO with Looplets[Ahrens,23]
- Piecewise auto-vectorisation[Augustine,19], [Pouchet,23]

# Approach



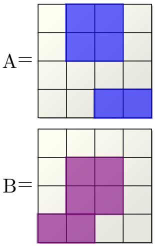

- **Compile part:** Abstraction of statements as equations.
- **Runtime part :** Propagation via evaluation.

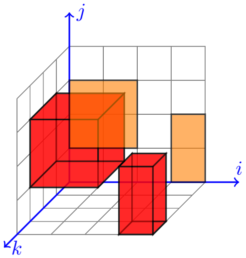
# Example

## Interest Regions

We define the interest regions of an expression $e$:
$[\![e]\!] = \{i \mid e(i) \neq 0\}$.

$$C[i,j] = \sum_{k=0}^{N-1} A[i,k] * B[k,j]$$

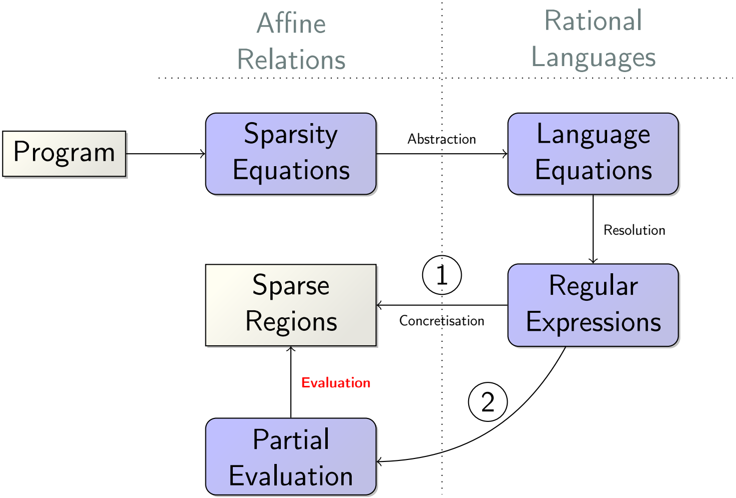



$[\![C]\!]$ is obtained by projecting non-zero computations with $(i,j,k) \to (i,j)$

# Plan

# Overview


Affine Relations
Rational Languages
Program
Sparsity Equations
Abstraction
Language Equations
Resolution
1
Sparse Regions
Concretisation
Regular Expressions
Evaluation
2
Partial Evaluation

# Intermediate Representation

Input dense code:

```
for t : 0..STEP
  for i: M..N
    for k: −M...M
      A[t,i]+=A[t−1,i+k];
```

Is translated to a **System of Affine Recurrence Equations**:

**SARE representation**

$$S[t,i] = \begin{cases} \sum_{k=-M}^{M} S[t-1, i+k] & t \geqslant 1 \\ \sum_{k=-M}^{M} \mathtt{In}[i+k] & t = 0 \end{cases}$$

- Dynamic assignment form.
- Code's dataflow.
- Automatic translation programs $\mapsto$ SARE available.

# Step 1: Sparsity Equations

**Equation's goals:**

- Equations propagate sparsity (polyhedra) across data-flow dependences (affine relations).
- $+ \rightarrow$ Union, $\times \rightarrow$ Intersection.

$$S[t,i] = \begin{cases} \sum_{k=-M}^{M} S[t-1, i+k] & t \geqslant 1 \\ \sum_{k=-M}^{M} \mathtt{In}[i+k] & t = 0 \end{cases}$$

$\Downarrow$ *Sparsity Equations*

$$\begin{aligned} [\![S]\!] &= [\![S]\!].\{(t-1, i+k) \rightarrow (t,i) : t \geqslant 1, -M \leqslant k \leqslant M\} \\ &\cup [\![\mathtt{In}]\!].\{(i+k) \rightarrow (t,i) : t = 0, -M \leqslant k \leqslant M\} \end{aligned}$$

# Step 2: Abstraction as regular expressions

**Rephrase with regular expression:**

- Relations and input polyhedra → Letter
- Composition → Concatenation
- Union → langage union

$$\begin{aligned} [\![S]\!] &= [\![S]\!].\{(t-1, i+k) \to (t, i) : t \geqslant 1, -M \leqslant k \leqslant M\} \\ &\cup [\![\mathtt{In}]\!].\{(i+k) \to (t, i) : t = 0, -M \leqslant k \leqslant M\} \end{aligned}$$

$\Downarrow$ *Abstraction*

$$L_S = L_S.a + L_{\mathtt{In}}.b$$

# Step 3&4: Resolution and concretization

$$L_S = L_S.a + L_{\mathtt{In}}.b$$

$\Downarrow$ *Resolution*

$$L_S = L_{\mathtt{In}}.b.a^*$$

$\Downarrow$ *Concretisation*

$$[\![S]\!] = [\![\mathtt{In}]\!].\{\ldots\}.\{\ldots\}^*$$

# Compile time: Evaluation

Two cases can occur when evaluating the system:

**No cycles ($L_S \rightarrow L_{In}$)**

Direct Evaluation ($L_S$, then $L_{In}$).

**Cycle remains ($L_S \leftrightarrow L_{In}$)**

Evaluate the cycle until a fix-point is met.

# Plan

# Context

Runtime evaluation.

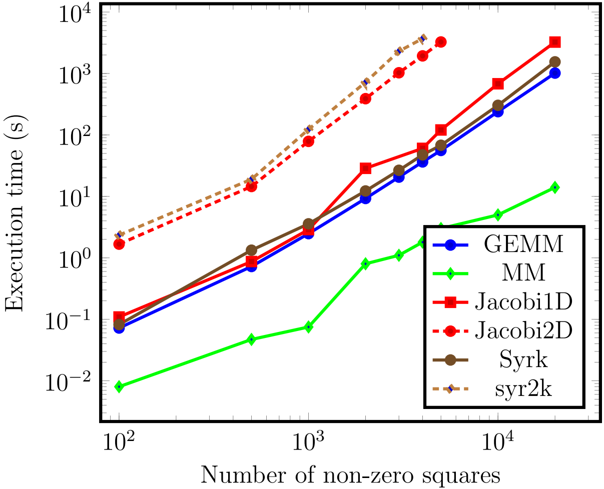
**Bench:**

- Sparse Matrix/Dense Matrix Multiply (MM)
- GEMM
- Syrk
- Syr2k
- Jacobi1d (J1D)
- Jacobi2d (J2D)

We use iscc for evaluation. Sparse inputs are represented as unions of polyhedra.

**CPU :** Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz

# Results




- bottlenecks are unions and intersections.
- ISL has trouble dealing with this much constraints.

# Plan

# Conclusion

**Our contributions:**

- We presented a static analysis for automatic specialization of dense polyhedral codes over sparse inputs.
- Experimental results show that bottlenecks are highly parallelizable.

**Future work:**

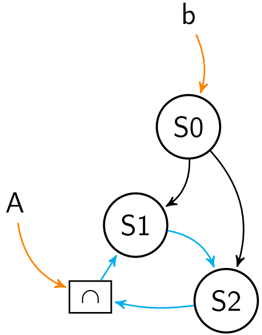
- Improve the evaluation step.
- Still rooms for system simplification.
- Can we get definitly get rid of cycles ?

Questions ?

```
for(i=0; i<N; i++)
    {
S0:     x[i]=b[i];
        for(j=0; j<i; j++)
S1:         x[i] = x[i] - A[i][j]*x[j];
S2      x[i] = x[i] / A[i][i];
}
```

$$\Downarrow$$

$$\begin{cases} [\![S_0]\!] = [\![b]\!] \\ [\![S_1]\!] = \left([\![S_0]\!].c \cup \left([\![A]\!] \cap [\![S_2]\!].r_i\right)\right).s_1{}^* \\ [\![S_2]\!] = [\![S_0]\!].d \cup [\![S_1]\!].e \end{cases}$$




- An intersection occur in the cycle $[\![S_1]\!] \leftrightarrow [\![S_2]\!]$
- **Partial evaluation**, which involves recurrence until fix-point.

## Construction rules

$$
\begin{array}{l}
[\![\texttt{array}]\!] = \{\texttt{[]} \mapsto \texttt{i} \mid \texttt{array(i)} \neq 0\} \\
[\![\texttt{constant}]\!] = \{\texttt{i} \mid \texttt{true}\} \ \textbf{if} \ \texttt{constant} \neq 0 \\
\qquad\qquad\qquad\ \ \emptyset \qquad\qquad\ \ \textbf{if} \ \texttt{constant} = 0 \\
[\![\texttt{array(u(i))}]\!] = [\![\texttt{array}]\!] \cdot \{\texttt{u(i)} \mapsto \texttt{i}\} \\
[\![\textbf{if}\texttt{(cond(i))} \ \textbf{then} \ \texttt{Expr1} \ \textbf{else} \ \texttt{Expr2}]\!] = \\
\qquad [\![\texttt{Expr1}]\!] \cdot \{\texttt{i} \mapsto \texttt{i} \mid \texttt{cond(i)}\} \cup \\
\qquad [\![\texttt{Expr2}]\!] \cdot \{\texttt{i} \mapsto \texttt{i} \mid \neg \ \texttt{cond(i)}\} \\
[\![\texttt{Expr1+Expr2}]\!] = [\![\texttt{Expr1}]\!] \cup [\![\texttt{Expr2}]\!] \\
[\![\texttt{Expr1*Expr2}]\!] = [\![\texttt{Expr1}]\!] \cap [\![\texttt{Expr2}]\!]
\end{array}
$$

**Entrée :** Système d'équations $(C_1 = E_1, \ldots, C_n = E_n)$
**Sortie :** Système d'équations équivalent à celui en entrée mais plus facile à manipuler pour la suite
$Resolution(C_1 = E_1, \ldots, C_n = E_n)$

```
for all i = n, n − 1, ..., 1 do
  Determiner F une formule ne dépendant que de C_1, ..., C_{i−1} par lemme d'Arden telle que C_i = F
  for all j ∈ ⟦1; i − 1⟧ do
    Substitution de C_i par F dans E_j
  end for
end for
```

## Syntax:

```
Sare ::= Statement+
Statement ::= array := Expr
Expr ::= i ↦ constant | array(i ↦ u(i))
       | if(i ↦ cond) then Expr else Expr
       | Expr + Expr | Expr × Expr | ...
```

## Functional form:

```
S := (i,j,k) ↦ \left(if(k<0) then 0
               else S(i,j,k−1)\right) + A(i,k) × B↩
                  (k,j)
C := (i,j) ↦ S(i,j,P−1)
```

**Syntax:**

```
Sare ::= Statement+
Statement ::= array := Expr
Expr ::= i ↦ constant | array(i ↦ u(i))
       | if(i ↦ cond) then Expr else Expr
       | Expr + Expr | Expr × Expr | ...
```

**Functional form:**

```
S := (i,j,k) ↦ \left(if(k<0) then 0
               else S(i,j,k−1)\right) + A(i,k) × B↩
                    (k,j)
C := (i,j) ↦ S(i,j,P−1)
```

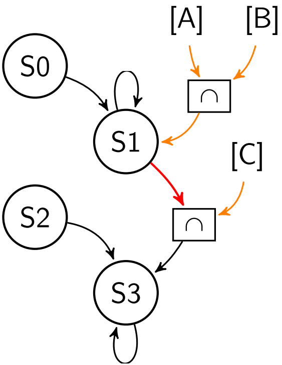
**Context:**

```
S := if((i,j,k) ↦ k<0) then (i,j,k) ↦ 0
     else (i,j,k) ↦ S(i,j,k−1)) +
          ((i,j,k) ↦ A(i,k) ×
           (i,j,k) ↦ B(k,j))
C := (i,j) ↦ S(i,j,P−1)
```

```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
S0:   T[i,j] = 0;
    for (int k=0; k<N; k++)
S1:     T[i,j] += A[i,k]*B[k,j];
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
S2:   R[i,j] = 0
    for (int k=0; k<N; k++)
S3:     R[i,j] += T[i,k]*C[k,j];
```

$$\Downarrow$$

$$\begin{cases} \mathsf{L}_{S0} = \mathit{false} \\ \mathsf{L}_{S1} = L_{S1}.s_1 \cup L_{S_0}.d \cup (a \cap b) \\ \mathsf{L}_{S2} = \mathit{false} \\ \mathsf{L}_{S3} = L_{S3}.s_3 \cup L_{S_2}.e \cup (L_{S1}.r \cap c) \end{cases}$$



# Example : trisolve

```
for(i=0; i<N; i++)
    {
S0:     x[i]=b[i];
        for(j=0; j<i; j++)
S1:         x[i] = x[i] - A[i][j]*x[j];
S2      x[i] = x[i] / A[i][i];
}
```

$$\Downarrow$$

$$\begin{cases} \mathrm{L}_{S0} = b \\ \mathrm{L}_{S1} = L_{S1}.s_1 \cup L_{S_0}.c \cup (a \cap L_{S_2}.r_i) \\ \mathrm{L}_{S2} = L_{S_0}.d \cup l_{S_1}.e \end{cases}$$