RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

Automatic Specialization of Polyhedral Programs on Sparse Structure

WP2 ExaSoft

Alec Sadler

September 24-26, 2025

Advised by **Christophe Alias**, INRIA Lyon

My thesis

Title: Specialization-based Dynamic Parallelization of Sparse Code.
→ WP2 - Task 2.3.

Main goal: Apply polyhedral transformation on sparse programs.

Our Approach:

- Data flow analysis on **general** polyhedral programs manipulating sparse inputs.
  → Automatic Dense/Sparse versioning of sub-domains.
  → Iteration space pruning.
  → Provide metrics for the runtime system.

Sparsity challenges

Polyhedral programs (SCoP)

Key Characteristics

- Affine loop-bounds $D = \{x \mid x \in \mathbb{Z}^n,\ Ax + BN + v \geq 0\}$.
- No aliases: (*T = *S) ✗.
- No Complex Control Flow: **No goto, break**.
- Data independent: if (A[i] != 0) ✗

SCoP ✓

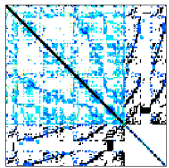
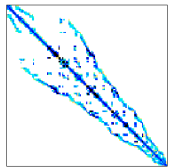
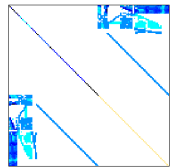
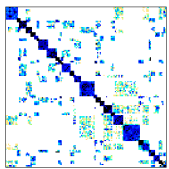
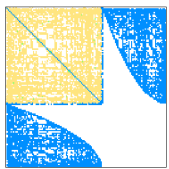
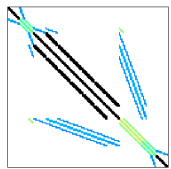
```
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    A[i][j] = B[i][j]+C[i][j];
  }
}
```

Not SCoP ✗

```
while (x < N) {
  if (A[x] == 0)
       {x = x / 2;}
  else { x = 3 * x + 1;}
}
```

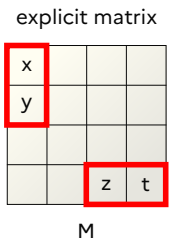
Data analysis on sparse programs

- Sparse matrices appear in pruned ML models, and scientific computing.
- They are too large $\rightarrow$ store only non-zeros!

Data analysis on sparse programs

- Sparse matrices appear in pruned ML models, and scientific computing.
- They are too large $\rightarrow$ store only non-zeros!



CSR format

ind	0	1	2	2	4

col	0	0	2	3
data	x	y	z	t

$\forall\, \mathsf{ind}[i] \leq j < \mathsf{ind}[i+1],$
$\mathsf{M}_{i,\mathsf{col}[j]} = \mathsf{data}[j]$

Sparsity makes data-analysis tricky

Dense matrix multiplication

```
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];
```

TACO (state of the art) SpGEMM

```
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 imbalance.
- Create non-convex iteration space (very tricky for polyhedral compilers!).

Dynamic data:

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

How can polyhedral code optimizations be used in a sparse context?

State of the art

State of the art

Sparse code generation:

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

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])

State of the art

Sparse code generation:

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

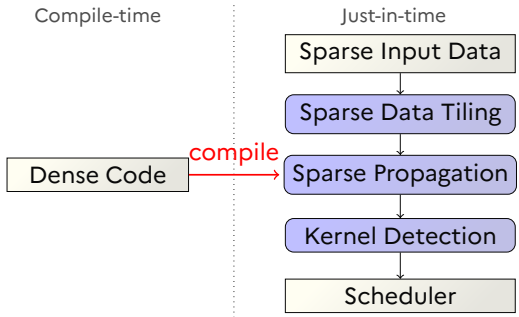
Sparse code optimization:

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

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

- DSL specialisation on "cycle programs" (Sympiler [Cheshmi,17], Parsy [Cheshmi,18])
- Specialization in TACO with Looplets [Ahrens,23]
- Piecewise auto-vectorisation [Augustine,19], [Pouchet,23]

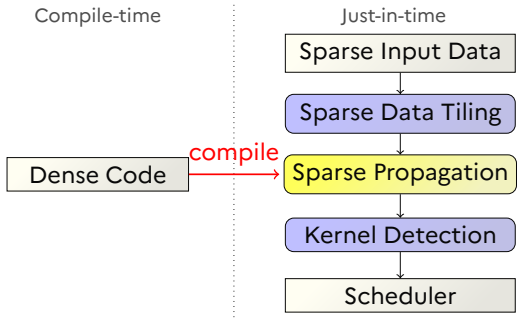
Project view



Key steps

- **Sparse Propagation** → Specialization.
- **Task recognition** → Map domains with BLAS kernels.

Project view



Focus: Sparse propagation

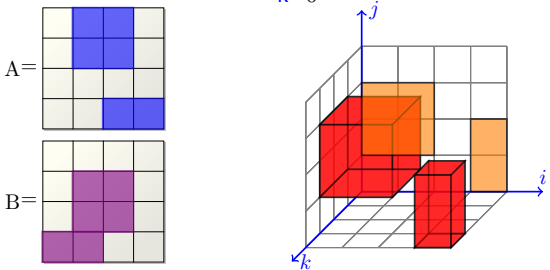
- **Compile time:** Abstraction of statements as equations.
- **Just-in-time:** Propagation via evaluation.

Example

Interest Regions

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

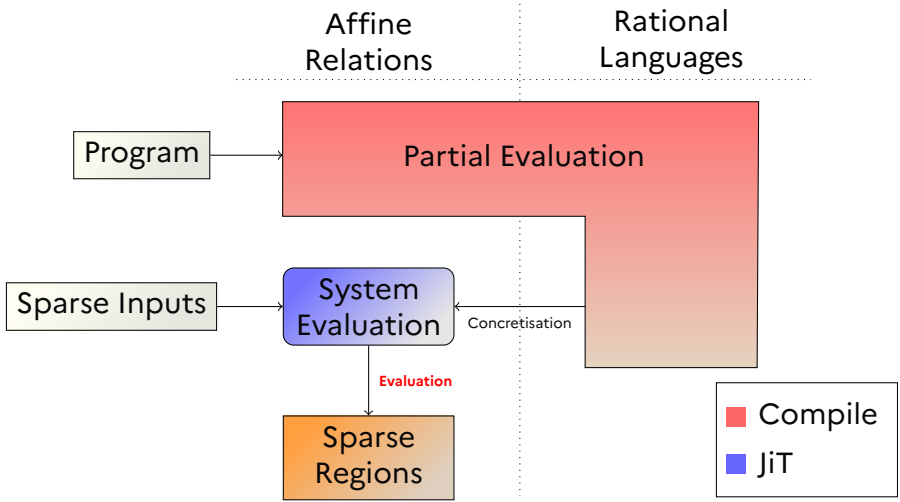
$$C[i,j] = \sum_{k=0}^{N-1} A[i,k] * B[k,j]$$



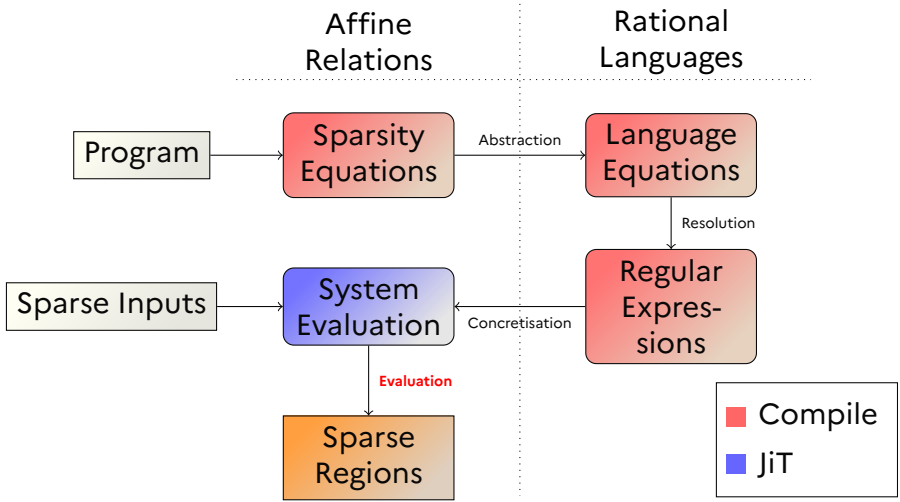
$[\![C]\!]$ is obtained by projecting non-zero computations with $(i,j,k) \rightarrow (i,j)$

Our approach

Overview


Affine Relations
Rational Languages
Program
Partial Evaluation
Sparse Inputs
System Evaluation
Concretisation
Evaluation
Sparse Regions
Compile
JiT

Overview


Affine Relations
Rational Languages
Program
Sparsity Equations
Abstraction
Language Equations
Resolution
Regular Expres-sions
Sparse Inputs
System Evaluation
Concretisation
Evaluation
Sparse Regions
Compile
JiT

Step 1: Sparsity Equations

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

$$\Downarrow \text{ Sparsity Equations}$$

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

Equation's goals:

- Equations propagate sparsity (polyhedra) across data-flow dependences (affine relations).
- Operations on SAREs become unions/intersections.

Step 2: Abstraction as regular expressions

$$[\![S]\!] = [\![S]\!].\{(t-1, i+k) \to (t, i) : t \geq 1, -M \leq k \leq M\}$$
$$\cup\, [\![In]\!].\{(i+k) \to (t, i) : t = 0, -M \leq k \leq M\}$$

$$\Downarrow \text{Abstraction}$$

$$L_S = L_S.a + b$$

Rephrase with regular expression:

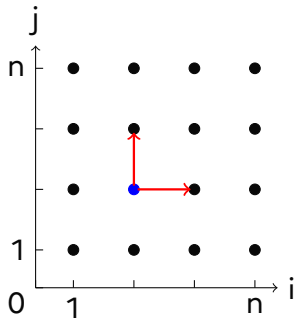
- Relations and input polyhedra $\to$ Letter
- Composition $\to$ Concatenation
- Union $\to$ langage union

Step 3: Resolution

$$L_S = L_S.a + b$$

$$\Downarrow \text{Resolution}$$

$$L_S = b.a^*$$

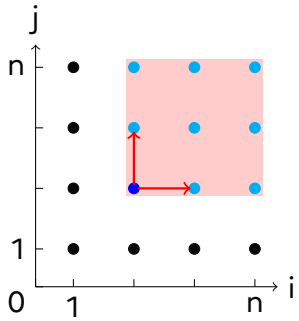


Step 3: Resolution

$$L_S = L_S.a + b$$

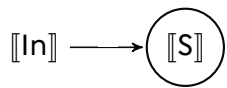
$$\Downarrow \text{Resolution}$$

$$L_S = b.a^*$$



Final step: Evaluation

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



We compute equations **along dependences**.

Special case: cycles in the system

Evaluate the cycle until a fix-point is met.
How to effectively evaluate the cycle ?

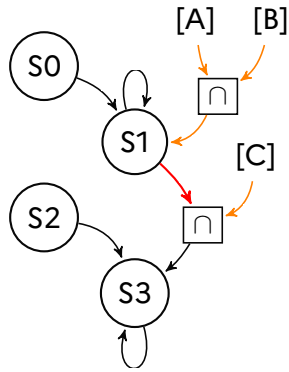
$\rightarrow$ Elimination tree, dependence graph, ...

Example with no cycles : A × B × C

```
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];
```

$$\Downarrow$$

$$\begin{cases} [\![\mathsf{S0}]\!] = \emptyset \\ [\![\mathsf{S1}]\!] = [\![\mathsf{S0}]\!] \cup (\mathsf{a} \cap \mathsf{b}).\mathsf{s}_1^* \\ [\![\mathsf{S2}]\!] = \emptyset \\ [\![\mathsf{S3}]\!] = [\![\mathsf{S2}]\!] \cup ([\![\mathsf{S1}]\!].\mathsf{r} \cap \mathsf{c}).\mathsf{s}_3^* \end{cases}$$



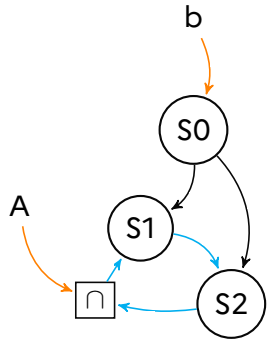
These equations are easily composable, here $[\![\mathsf{S}_1]\!] = [\![\mathsf{T}]\!]$ and $[\![\mathsf{S}_3]\!] = [\![\mathsf{R}]\!]$.

Example with cycles: Triangular solve

```
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];
}
```

$$\Downarrow$$

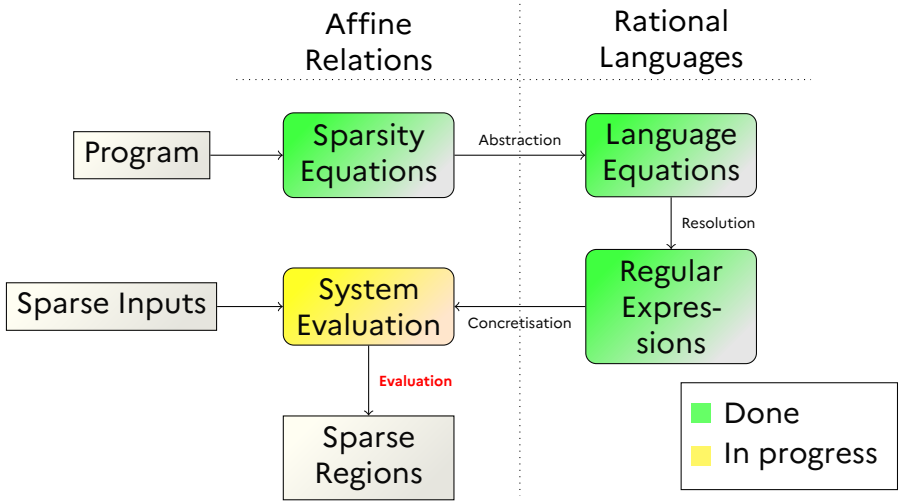
$$[\![S_0]\!] = [\![b]\!]$$
$$[\![S_1]\!] = ([\![A]\!] \cap [\![S_2]\!].\{j \to (i,j)\}.\{(i,j) \to i\})^*$$
$$[\![S_2]\!] = [\![S_1]\!].\{i \to i\} \cup [\![S_2]\!]$$



- We can't precompute S1.
- This iterative process is equivalent to a BFS of DG_A.

Experimental results

Overview

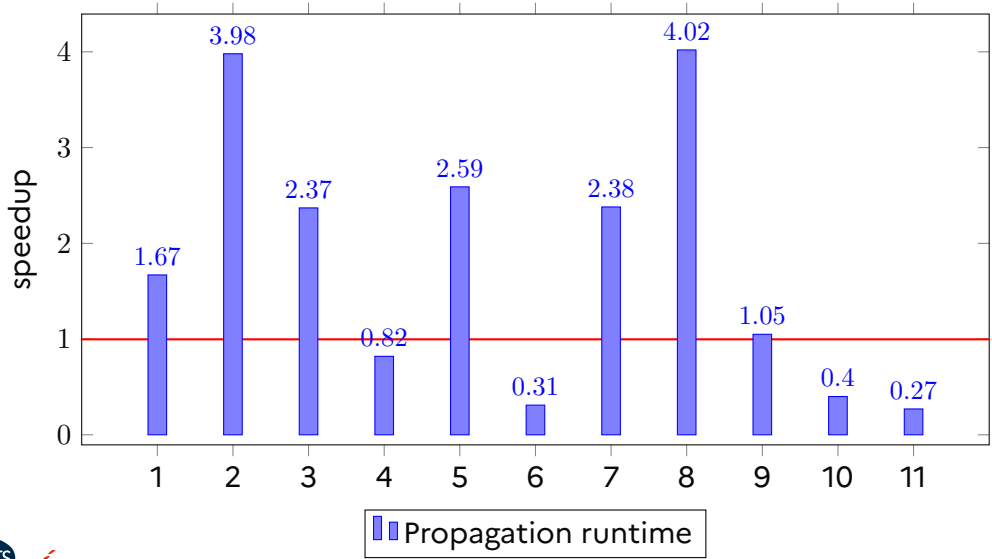

Affine Relations
Rational Languages
Program
Sparsity Equations
Abstraction
Language Equations
Resolution
Regular Expres-sions
Sparse Inputs
System Evaluation
Concretisation
Evaluation
Sparse Regions
Done
In progress

Experimental setup

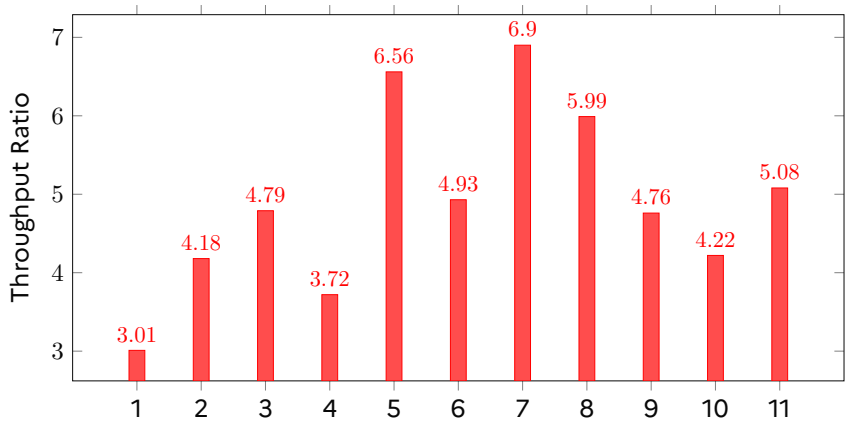
- Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 6 cores.
- Generation of templated parallel code, not fully automated.
- Baseline: TACO assemble method.
- Kernels: SPGEMM and SPGEMM^2(A$\times$B & A $\times$ B $\times$ C).

Matrix name(ID)	N (10^3)	nnz (10^6)	kind
C-50(1)	22.4	0.180	Optimization
LowThrust_12(2)	18.4	0.224	Optimal Control
Rajat26(3)	51.0	0.247	Circuit Simulation
Gyro_m(4)	17.3	0.340	Model Reduction
Thermal1(5)	82.6	0.574	Thermal
Pres_poisson(6)	14.8	0.715	Fluid Dynamic
Brack2(7)	62.0	0.733	2D Problem
Ma2010(8)	157	0.766	Undirected Graph
Dubcova2(9)	65.0	1.03	2D Problem
SiO(10)	33.4	1.31	Quantum Chemistry
Pkustk06(11)	43.1	2.57	Structural

Benchmarks - Speedups



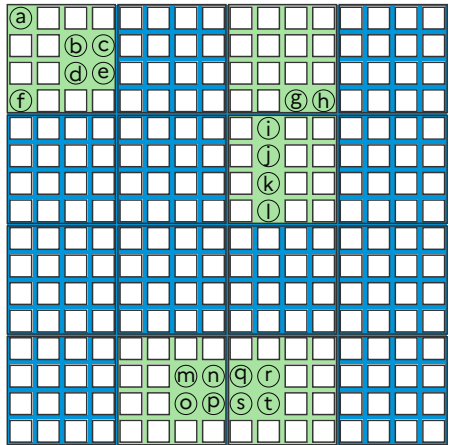
Benchmarks - Throughput



Throughput = Iteration Processed Per Second.
→ **4.9x** speedup in average.

Future Work

- Further simplification of sparsity equations.
- Recognize Data dependence patterns in direct solver kernels (etree).
- Code Generation with data tiling.



Questions?