Polyhedral Compilation

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Polyhedral Compilation

- General technique for optimizing parts of code that have 'dynamic instances' of instructions
  - Dynamic instance = think of instructions that are part of loop body
  - includes analysis and transformation

- The phrase began to be used from around 2006 by Sylvain Girbal and others in their paper titled "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies"

- Makes use of Polyhedra – a mathematical formalism. Alternatives: abstract interpretation and array region analysis

Polyhedral Compilation – use case

```
for(i=0;i<3;i++)
S:  B[i] = foo(A[i])

for(i=0;i<3;i++)
T:  C[i] = bar(B[2-i])
```

$$\{\, S[i]\,, T[i] \,\}$$

$$\{\, S[i] \rightarrow [i] \,\} \qquad \{\, T[i] \rightarrow [i] \,\}$$

```
S[0]        S[1]        S[2]
R: A[0]     R: A[1]     R: A[2]
W: B[0]     W: B[1]     W: B[2]

T[0]        T[1]        T[2]
R: B[2]     R: B[1]     R: B[0]
W: C[0]     W: C[1]     W: C[2]
```

$$\{\, S[0]\; T[2], S[1]\; T[1], S[2]\; T[0] \}$$

$$S[\quad], T[\quad]$$

$$\{\, S[i] \rightarrow [i]\;;\; T[i] \rightarrow [2-i] \}$$

$$\{S[i]\}, \{T[i]\}$$

```
for(j=0;j<3;j++){
S:  B[j] = foo(A[j])
T:  C[j] = bar(B[2-j])
}
```

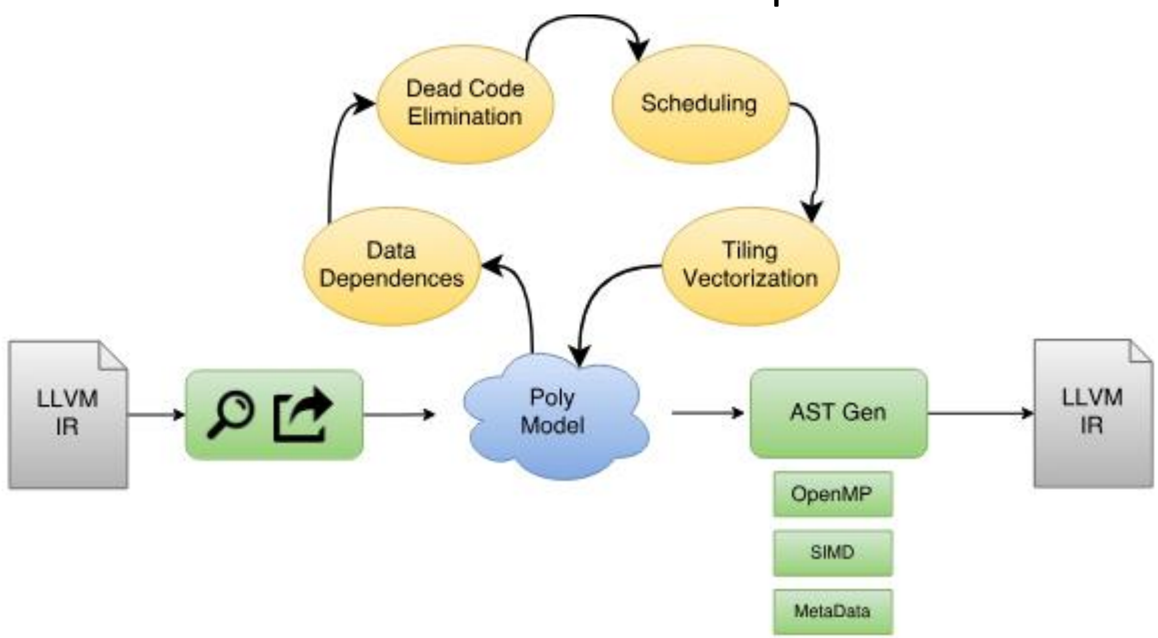
- Loop Fusion

Polyhedral Compilation - Terminology

- Instance Set
  - Elements are named integer tuples e.g., T[0]
- Dependence Relation
  - Expressed as mapping from set elements to data that those elements access
  - Read,Write,May-Read,May-Write,Must-write, RAW, WAR, WAW
- Schedule
  - Strict partial order on the elements of the set. One way to represent is the schedule tree.

Polly - Polyhedral Compilation in LLVM

- Frontend
  - Convert LLVM IR to polyhedral representation
- Middle-end
  - Analyze polyhedral representation and transform (model) if feasible
- Backend
  - Generate LLVM IR from transformed representation



source: https://llvm.org/devmtg/2016-03/Tutorials/applied-polyhedral-compilation.pdf

IR to Polyhedral Representation

- Step1: Identify relevant parts of code and create polyhedral representation. Starting point: LLVM IR
- SCoP – static control parts (mostly loops)
  - Single induction variable incremented from lower value to higher. Stride one.
  - The following are affine expressions:
    - Upper and lower bounds of loops
    - Parameters: parameter is any integer variable that is not modified inside the SCoP.
    - Array subscripts involving induction vars and parameters
  - Only valid statements are assignments that store the result of an expression to an array element. The expression involving induction vars, parameters, or array elements is side-effect free.
  - How to detect SCoP? – Use simple loop detection pass, Dominance Info (how?)

Affine Expressions

- Simply put, expressions that have addition but no multiplication (if multiplication, then constant is okay).
- Kind of linear equation
- E.g. `i+20`, `2-i`, `10*n`, etc. but not i*j, i*i, n*n etc.

SCoP

- Challenges when the starting point is LLVM IR
  - Recovering expressions for loop bounds
  - Array subscript expressions
  - Conditions with if (yes, we can have if stmt inside loop body)
    - Any structured code is okay for SCoP

Why? At IR level, the optimizations may have hoisted invariants, performed strength reduction, etc.

Answer: SCEV: recover closed-form expression of all scalar integer variables. Recall: SCEV outputs chain of recurrence.

Analyze chain of recurrence to check if an expression is affine

SCoP

- Example SCoPs
  - for loops, while loops, do-while loops with/without pointer arithmetic, etc.

```
int i = 0;
do {
   int b = 2 * i;
   int c = b * 3 + 5 * i;
   A[c] = i;
   i += 2;
} while (i < N);
```

```
int A[1024];
int *B = A;
while (B < &A[1024]) {
   *B = 1;
   ++B;
}
```

Polyhedral Representation

- Step2: represent SCoP with the help of
  - Domains
    - Domain of S is {[0], [1], [2]}

```
for(i=0;i<3;i++)
S:  B[i] = foo(A[i])
```

    - Exercise: what is the domain of T?

```
for(i=0;i<n+m;i++)
T:  B[i] = foo(A[i])
```

  - Schedule
    - Is a relation when applied on the domain, tells at what time the computation is performed. E.g. $S[i] \rightarrow [i]$; //S is executed at time i
  - Memory accesses

- integer set library (isl)
  - Tool used for operating on Presburger sets and relations
  - Domains are represented using Presburger formulas.
  - Presburger arithmetic: first-order logic over natural numbers
- Apply polyhedral transformations (if required)
- Export and import of schedules

Polyhedral Representation to IR

- Step3: Polyhedral representation to LLVM IR
  - First an AST is produced, from the AST LLVM IR is produced.

- If loops are parallelizable, the following codes can be generated.
  - SIMD code
  - OMP Parallel Code

Polyhedral Representation

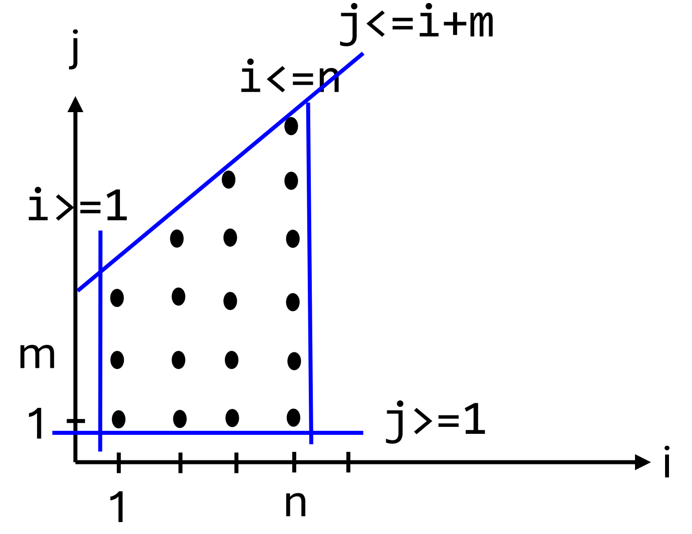
```
for(i=1;i<=n;i++)
    for(j=1;j<=i+m;j++)
        S: foo(i,j)
```

<u>Execution Instances</u>

state: n=4, m=2

			foo(4,6)
		foo(3,5)	foo(4,5)
	foo(2,4)	foo(3,4)	foo(4,4)
foo(1,3)	foo(2,3)	foo(3,3)	foo(4,3)
foo(1,2)	foo(2,2)	foo(3,2)	foo(4,2)
foo(1,1)	foo(2,1)	foo(3,1)	foo(4,1)

<u>Iteration space</u>



$$\{S(i,j)\ |1 \le i \le n \land 1 \le j \le i+m\}$$

Polyhedral Representation

```
for(i=1;i<=n;i++)
    for(j=1;j<=i+m;j++)
        S: foo(i,j)
```

- Model

$$I_S = \{S(i,j) \mid 1 \le i \le n \wedge 1 \le j \le i+m\}$$

$$\Theta_S = \{S(i,j) \rightarrow (i,j)\} \longleftarrow \text{schedule}$$

Loop interchange:

$$\Theta_S = \{S(i,j) \rightarrow (j,i)\}$$

Strip mining:

$$\Theta_S = \left\{S(i,j) \rightarrow (\left\lfloor \frac{i}{4} \right\rfloor, j, i\, mod\ 4)\right\}$$

```
for(i=1;i<=n;i++){
  a[i]=b[i]+c[i];
}
```

$\Longrightarrow$

```
for(i=1;i<=n;i++){
        for(j=1;j<=4;j++)
                a[i][j]=b[i][j]+c[i][j];

}
```

Polyhedral Representation

```
for(i=1;i<=n;i++)
    for(j=1;j<=i+m;j++)
        S: foo(i,j)
```

- Model

$$I_S = \{S(i,j) \mid 1 \le i \le n \land 1 \le j \le i+m\}$$

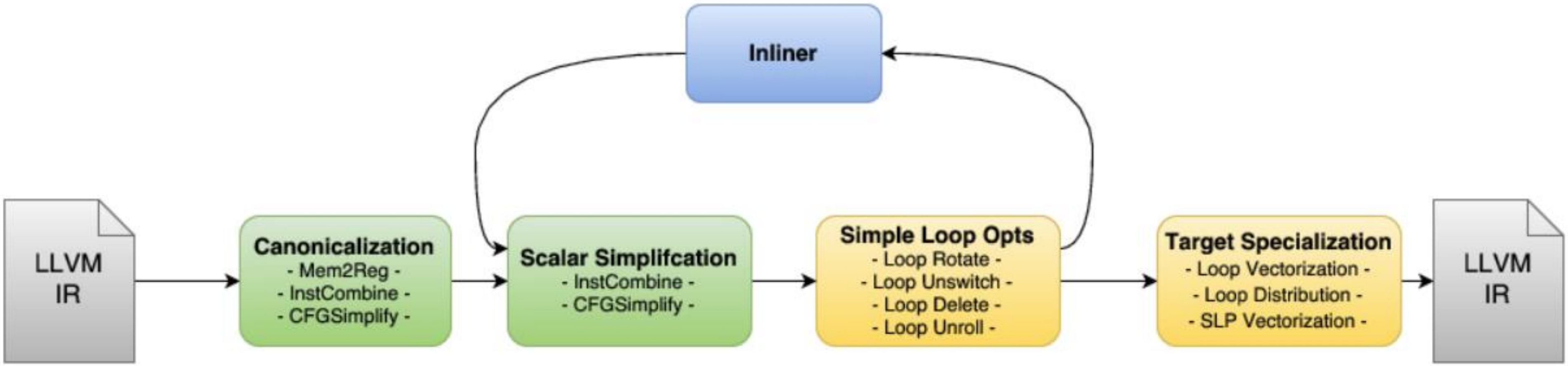
$$\Theta_S = \{S(i,j) \to (i,j)\} \longleftarrow \text{schedule}$$

tiling (4x4 tiles):

$$\Theta_S = \left\{S(i,j) \to (\left\lfloor \frac{i}{4} \right\rfloor, \left\lfloor \frac{j}{4} \right\rfloor, j \bmod 4, i \bmod 4)\right\}$$

Polly in LLVM's Pass Pipeline

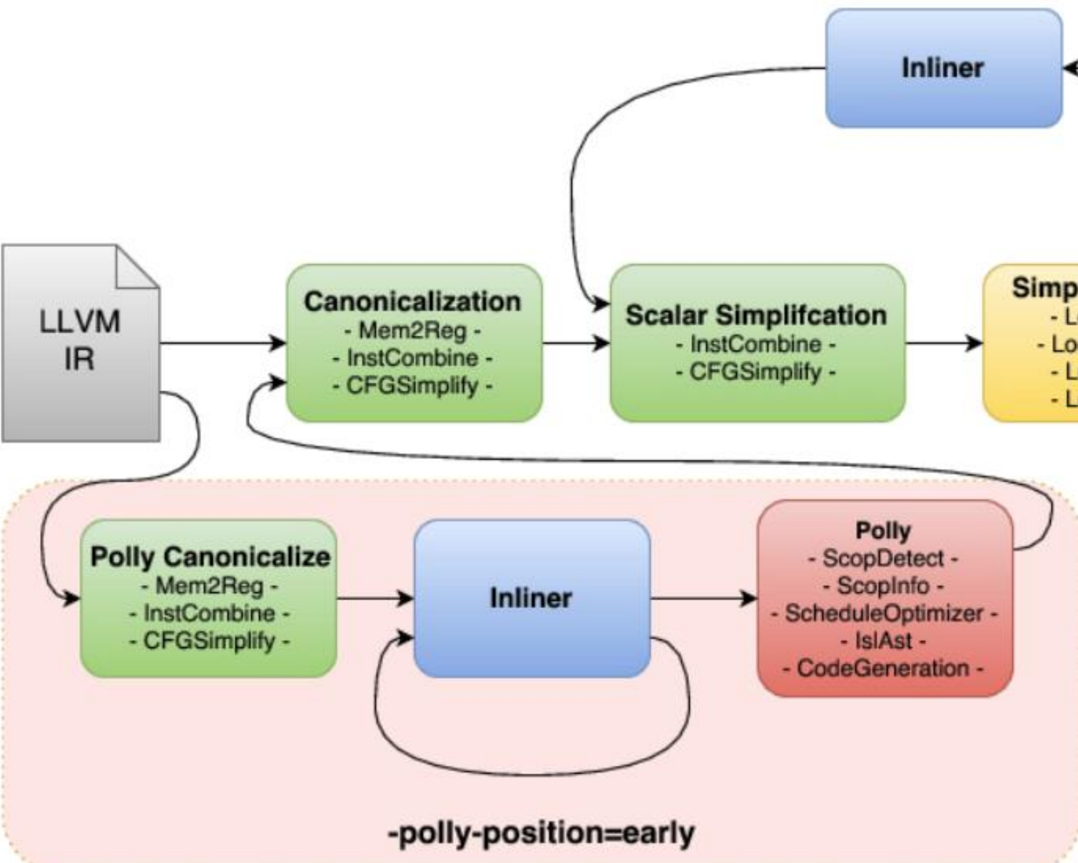
- Passes corresponding to Clang/Opt O1/O2/O3 consists of 3 broad categories



source: https://polly.llvm.org/docs/Architecture.html#polly-in-the-llvm-pass-pipeline

Polly in LLVM's Pass Pipeline

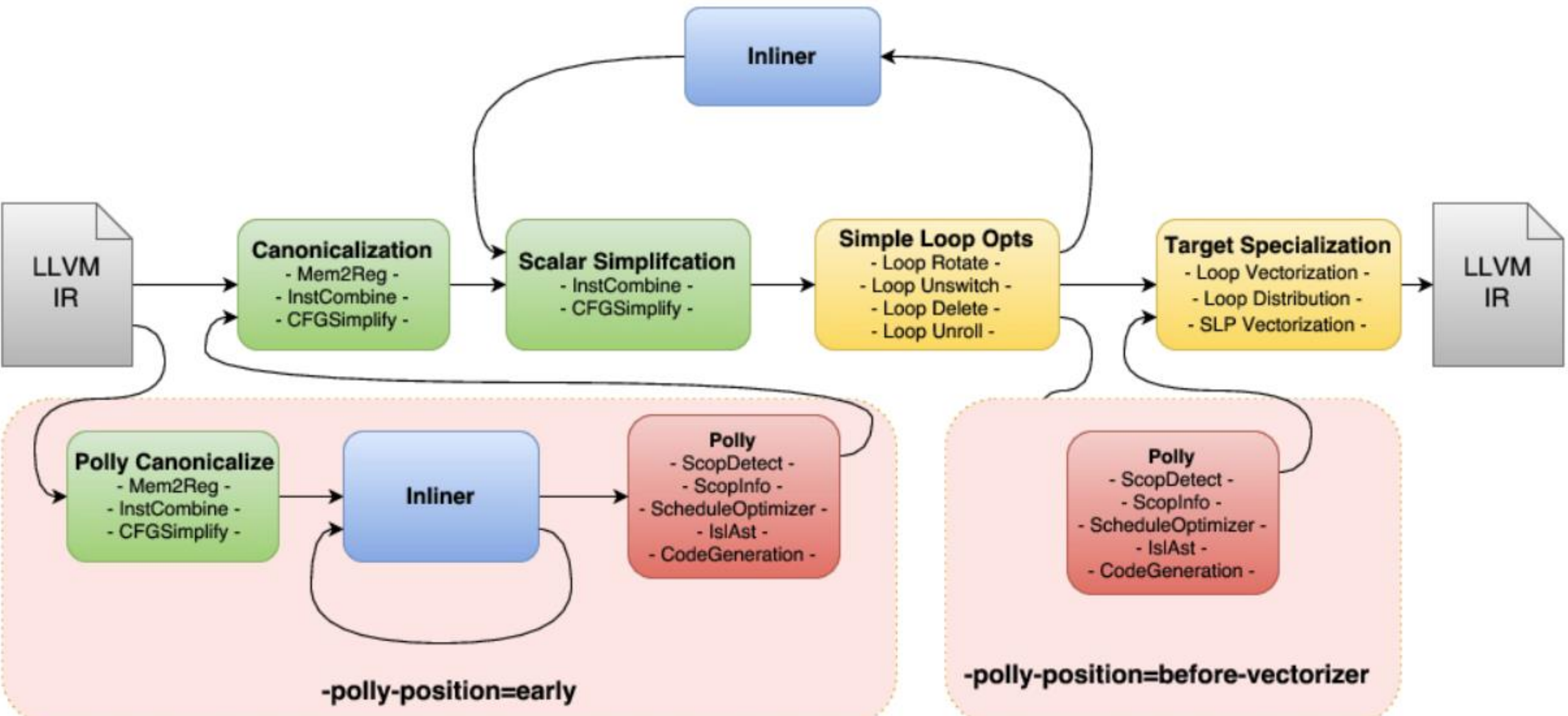
- Running Polly early



source: https://polly.llvm.org/docs/Architecture.html#polly-in-the-llvm-pass-pipeline

Polly in LLVM's Pass Pipeline

- Running Polly before vectorizer



source: https://polly.llvm.org/docs/Architecture.html#polly-in-the-llvm-pass-pipeline