Polyhedral Compilation - III

Nikhil Hegde

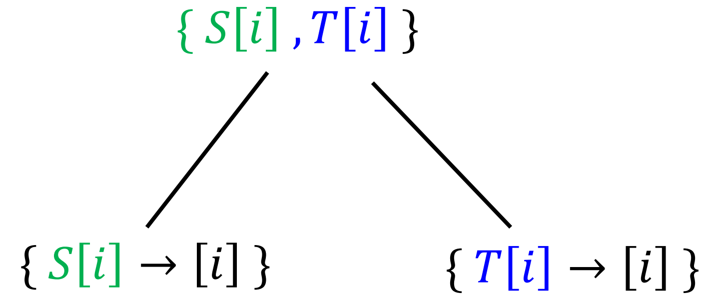
Compiler Optimizations in LLVM

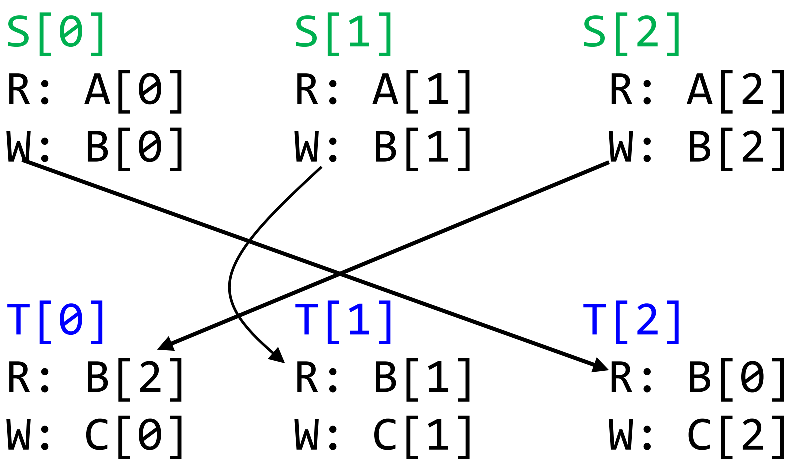
Lecture series @ QUALCOMM Inc.

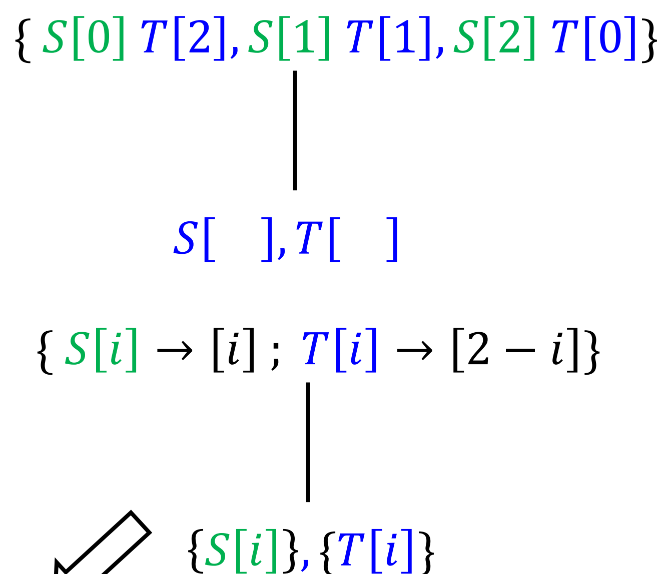
Recap: Polyhedral Compilation – use case

```
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])
```







- Loop Fusion

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

Recap: Polly

- SCoPs
  - SCoPs that don't exist, SCoPs that are and are not profitable,
  - Viewing Polly's activity using `-Rpass-analysis, –Rpass-missed`
  - Highlighting with `-dot-scops-only`
- Representation
  - Polly-scops : creates polyhedral description of SCoPs
- Optimization
  - Optimizing matrix multiplication with polly
- Saving the polyhedral representation in a file (export to jscop)
- Loading the saved representation into polly (import from jscop)
- Code generation based on jscop file loaded
- Target generation and performance comparison

Recap: Polyhedral Representation

Focus of today's class.

- Step2: represent SCoP with the help of
  - Domains
    - Domain of S is {[0], [1], [2]}
    - Exercise: what is the domain of T?
  - Schedule
    - Is a relation when applied on the domain, tells at what time the computation is performed. E.g. $S[i] \to [i]$; //S is executed at time i
  - Memory accesses

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

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

- 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

Can we reverse this loop?

```
for(i=1;i<=4;i++)
    S: A[i]=A[i-1]
```

⟹

```
for(i=4;i>=1;i--)
       A[i]=A[i-1]
```

time

S[1]	R: A[0] W: A[1]
S[2]	R: A[1] W: A[2]
S[3]	R: A[2] W: A[3]
S[4]	R: A[3] W: A[4]

time

S[4]	R: A[3] W: A[4]
S[3]	R: A[2] W: A[3]
S[2]	R: A[1] W: A[2]
S[1]	R: A[0] W: A[1]

S[i] is the Producer of data and S[i+1] is the Consumer of data

Is S[i] still the Producer of data and S[i+1] the Consumer of data?

Arrows can't go backwards in time!

- Are any dependencies violated?
  - We have turned flow dependencies (RAW) in the original program to anti-dependencies (WAR) in the modified program!

Mathematical Formulation

```
for(i=1;i<=4;i++)
    S: A[i]=A[i-1]
```

⇨

```
for(i=4;i>=1;i--)
       A[i]=A[i-1]
```

- Set Notation

S[1]
S[2]
S[3]
S[4]

{ S[i] | 1<=i<=4 }

S[4]
S[3]
S[2]
S[1]

{ S[i] | 1<=i<=4 }

  - Order of statements, S[i], that are executed i.e.,

  Schedule in polyhedral terminology is nothing but map of statement instances to time

  { S[i] -> i}

  { S[i] -> 5-i}

  - Set of data dependences: function of (schedules, memory access pattern)

  producer **consumer**

  { (S[i], S[i+1]) | 1<=i<=3}

Mathematical Formulation

```
for(i=1;i<=4;i++)
    S: A[i]=A[i-1]
```

⟹

```
for(i=4;i>=1;i--)
    A[i]=A[i-1]
```

- Transformation is legal if set of violated data dependencies is empty

S[1]
S[2] `{ S[i] | 1<=i<=4 }`
S[3]
S[4] `{ S[i] -> i}`

S[4]
S[3] `{ S[i] | 1<=i<=4 }`
S[2]
S[1] `{ S[i] -> 5-i}`

- Set of violated data dependencies in new schedule = Set of pairs of statements (a, b), where
  - Statement a sends data to b AND a comes after b in the new schedule

`{ (S[i], S[i+1]) | 1<=i<=3}` ⋂
`(S[i], S[j]) | newsched(i)` $\geq$ newsched(j)

`{ (S[i], S[i+1]) | 1<=i<=3}` ⋂
`newsched(i)` $\geq$ newsched(i + 1)

Mathematical Formulation

```
for(i=1;i<=4;i++)
  S: A[i]=A[i-1]
```

⇒

```
for(i=4;i>=1;i--)
     A[i]=A[i-1]
```

- Transformation is legal if set of violated data dependencies is empty

S[1]
S[2] { S[i] | 1<=i<=4 }
S[3]
S[4] { S[i] -> i}

S[4]
S[3] { S[i] | 1<=i<=4 }
S[2]
S[1] { S[i] -> 5-i}

- Set of violated data dependencies in new schedule = Set of pairs of statements (a, b), where
  - Statement a sends data to b AND a comes after b in the new schedule

{ (S[i], S[i+1]) | 1<=i<=3} $\cap$
(S[i], S[j]) | newsched(i) $\geq$ newsched(j)

{ (S[i], S[i+1]) | 1<=i<=3} $\cap$
newsched(5-i) $\geq$ newsched(5 − (i + 1))

Mathematical Formulation

```
for(i=1;i<=4;i++)
     S: A[i]=A[i-1]
```

⇨

```
for(i=4;i>=1;i--)
        A[i]=A[i-1]
```

- Check if the set is empty:

{ (S[i], S[i+1]) | 1<=i<=3 && $5\text{-i} \geq 5 - (\mathrm{i} + 1)$}

- The set is empty iff the system of linear equalities cannot be satisfied

1<=i<=3
5-i≥5-(i+1)

This is satisfiable for i=1
1<=1<=3   => 1<=3
5-1≥5-(1+1) => 4>=3

Integer Linear Programming (ILP)

- Is a tool to check if Check if system of constraints can be satisfied or not

  1<=i<=3

  5-i≥5-(i+1)

  - Solves linear integer equations and inequalities
  - Also optimizes linear objective functions
  - E.g. ILP problem:

    find x, y, z such that:
    3x+4y+5>=0
    -3x-3<=0
    x+z+2=0

    find x, y, z that minimize x+y+z
    3x+4y+5>=0
    -3x-3<=0
    x+z+2=0

  - Not ILP:

    3xy+4y+5 >=0
    for all x>=0 -3x-3<=0
    x+z+2=0

    3x+4y+5+**sin(x)** >=0
    -3x-3<=0
    x+z+2=0

    3**xy**+4y+5 >=0
    -3x-3<=0
    x+z+2=0

Integer Linear Programming (ILP)

- ILP problem is NP-complete in theory
- Tractable in practice for 100s of variables
- Can be used even when operators involved:
  - min, max, remainder, abs, division, remainder, etc.
  - Propositional logic

Mathematical Formulation

```
for(i=1;i<=4;i++)
    for(j=1;j<=3;j++)
        S:  A[i][j]=A[i-1][j+1]
```

- Schedule: `{ S[i,j] -> (i,j)}`

- How do we express order among set elements? i.e.

  `[i,j] > [i+2,j-2]`

  - We are dealing with vectors and no total order while comparing vectors
  - Express partial order with lexicographical ordering

  ```
  [i,j] > [i+2,j-2] iff (i > i+2) ||
                        ((i = i+2) && (j > j-2))

  e.g. [1,0] > [0,2]
  ```

  - ILP solver cannot directly handle ||. So, make multiple calls.

Loop Interchange – ILP Way

```
for(i=1;i<=4;i++)
  for(j=1;j<=3;j++)
    S:  A[i][j]=A[i-1][j+1]
```

⇒

```
for(j=1;j<=3;j++)
  for(i=1;i<=4;i++)
    S:  A[i][j]=A[i-1][j+1]
```

- Schedule: `{ S[i,j] -> (i,j)}` `{ S[i,j] -> (j,i)}`

```
[i' j'] < [j,i] &&
i' = 1+i &&
j' = -1+j &&
1<=i<=4 &&
1<=j<=3
```

Recall: statement a sends data to b AND a comes after b in the new schedule

- The above is satisfiable. So, the transformation is illegal.

How do we discover new transformations?

Polyhedral Compilation – Use cases

- So far we have seen Analysis (check legality)
  - Extract initial schedule and data dependences
  - Construct final schedule that we want
  - Check if final schedule breaks dependences (solve ILP)

- Transformation
  - Extract initial schedule and data dependences
  - Set objective function that captures what we want and constraints that capture all dependencies
  - Solve ILP

Loop Fusion – ILP Way

```
for(i=0;i<=5;i++)
    P:  A[i]=X[i]+1

for(j=0;j<=5;j++)
    C:  B[j]=A[j]*3
```

⇨

```
for(i=0;i<=5;i++)
    P:  A[i]=X[i]+1
    C:  B[i]=A[i]*3
```

- New schedule is an affine function of loop index variables in the original program

```
for(i=0;i<=5;i++)
    P:  A[i]=X[i]+1            SP(i)=sp*i+dp

for(j=0;j<=5;j++)
    C:  B[j]=A[j]*3            SC(j)=sc*j+dc
```

- sp, sc, dp, dc are schedule parameters
  - sp, sc define the rate at which instruction is executed. dp, dc define the initial wait time before the first instruction is executed.
- Optimization problem: pick values for sp, sc, dp, dc

Loop Fusion – ILP Way

- Optimization problem
  - Optimize a function that captures locality of modified schedules. Subject to:

```
for(i=0;i<=5;i++)
    P:  A[i]=X[i]+1

for(j=0;j<=5;j++)
    C:  B[j]=A[j]*3
```

for all i, j such that P(i) sends data to C(j). SP(i) <= SC(j)

for all i<=0<=5 && 0<=j<=5 && i=j. SP(i) <= SC(j)

for all i<=0<=5 && 0<=j<=5 && i=j. `sp*i+dp <= sc*j+dc`

  - But the above is not an ILP problem. Why?

Farkas Lemma

- An affine function is non-negative over a polyhedron iff it can be written as a non-negative combination of the constraints that that form the polyhedron.
  - Affine form of Farkas Lemma

for all x in {x | Ax +b > 0}. $s^T x+d >=0$ iff

there exist p0, p >=0. for all x. $s^T x+d = p^T (Ax+b)+p0$

E.g. for all x > 0. a*x>=0

Farkas Lemma: there exist p0, p1 >=0. for all x. a*x = p1*x+p0

Isolate "for all": there exist p0, p1 >=0. for all x. $\boxed{(a-p1)*x - p0 = 0}$

there exist p0, p1 >=0. (a-p1)=0 && p0 = 0

SAT problem: p0>=0 && p1 >=0 && (a-p1)=0 && p0 = 0

Loop Fusion – ILP Way

- Apply Farkas Lemma to the optimization problem:

  Optimize a function that captures locality of modified schedules.
  Subject to: for all i<=0<=5 && 0<=j<=5 && i=j. `sp*i+dp <= sc*j+dc`

```
for(i=0;i<=5;i++)
    P:  A[i]=X[i]+1

for(j=0;j<=5;j++)
    C:  B[j]=A[j]*3
```

for all i<=0<=5 && 0<=j<=5 && i=j. `sp*i+dp-sc*j–dc<=0`

for all i<=0<=5 && 0<=j<=5 && i=j. `sc*j-sp*i+dc-dp>=0`

$$\begin{bmatrix} sc \\ -sp \end{bmatrix} \quad \begin{bmatrix} i \\ j \end{bmatrix}$$

0<=i && i<=5 && 0<=j && j<=5 && i<=j && j<=i. `sc*j-sp*i+dc-dp>=0`

i>=0 && -i>=-5 && j>=0 && -j>=-5 && j-i>=0 && i-j>=0. `sc*j-sp*i+dc-dp>=0`

for all i<=0<=5 && 0<=j<=5 && i=j. Minimize w, where `sp*i+dp-sc*j–dc <=w`

Intuitively: w imposes a bound on the time between production and consumption of data

Creating Parallelism – ILP Way

- Apply Farkas Lemma to the optimization problem:

  $\boxed{\text{Optimize a function}}$ that captures locality of modified schedules.
  Subject to: for all i<=0<=5 && 0<=j<=5 && i=j. `sp*i+dp <= sc*j+dc`

for all i<=0<=5 && 0<=j<=5 && i=j. Maximize w, where w<=`sc*j+dc-sp*i-dp`
`0<=w<=MAX_BOUND`

Intuitively: schedule producers and consumers as far as possible

Feautrier's algorithm for extracting parallelism is based on similar idea.

Handling Multiple Dimensions

- Template

```
Initialize data dependence_graph
Initialize schedule_vector to be empty
while(dependence graph is not empty) {
        next_schedule = solve_ilp(dependence_graph, objective)
        schedule_vector.append(next_schedule)
        dependence_graph = remove_carried_dependency(next_schedule)
}
```

- Start from outermost loop

Generating Code

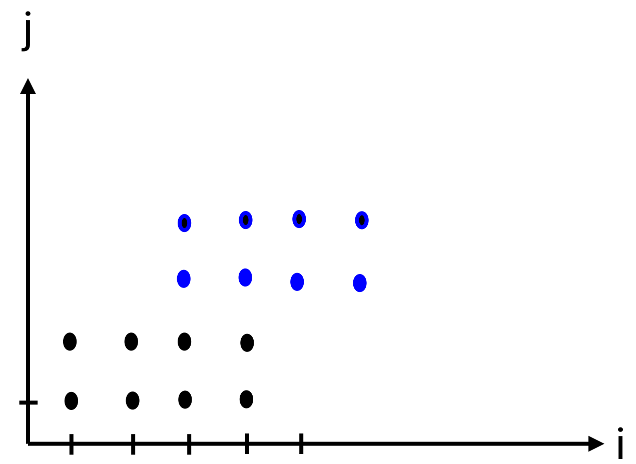
- Turning schedules back into for loops

  - After transformation, we have functions such as SP(i), SC(i) that map instances of statements to times in the new schedule

  - E.g. Two rectangular polyhedral

```
A = { [i,j] | 1<=i<=4 AND 1<=j<=2 }

B = { [i,j] | 3<=i<=6 AND 3<=j<=4 }
```



  - Walk over the points in lexicographic order

Generating Code

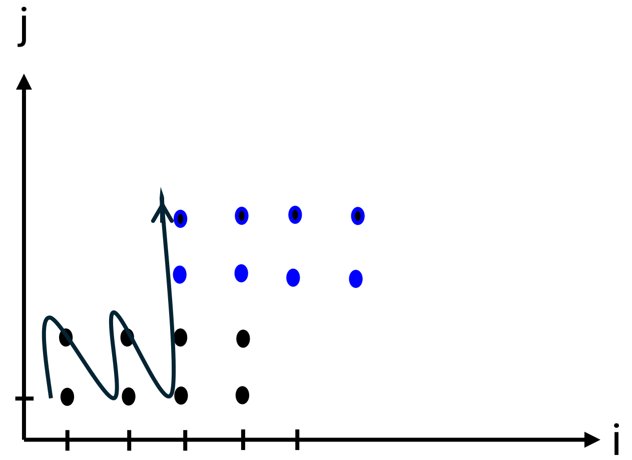
- Turning schedules back into for loops

  - After transformation, we have functions such as SP(i), SC(i) that map instances of statements to times in the new schedule

  - E.g. Two rectangular polyhedral

    ```
    A = { [i,j] | 1<=i<=4 AND 1<=j<=2 }

    B = { [i,j] | 3<=i<=6 AND 3<=j<=4 }
    ```

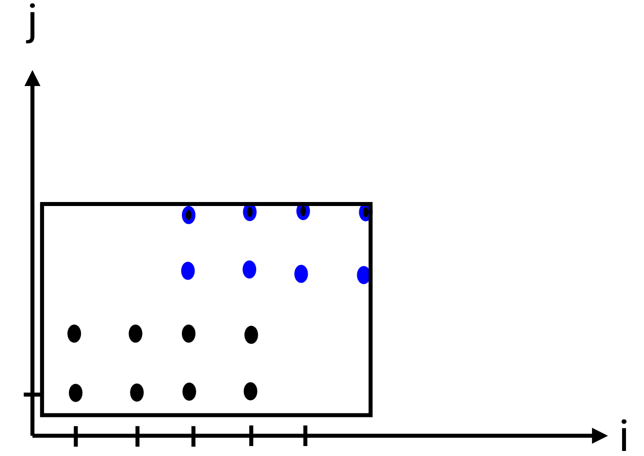


  - Walk over the points in lexicographic order. How do we generate for loops for these points?

Generating Code

- Turning schedules back into for loops
  - Compute a convex hull of statements
  - Generate a perfect loop nest
  - Use polyhedral as guards

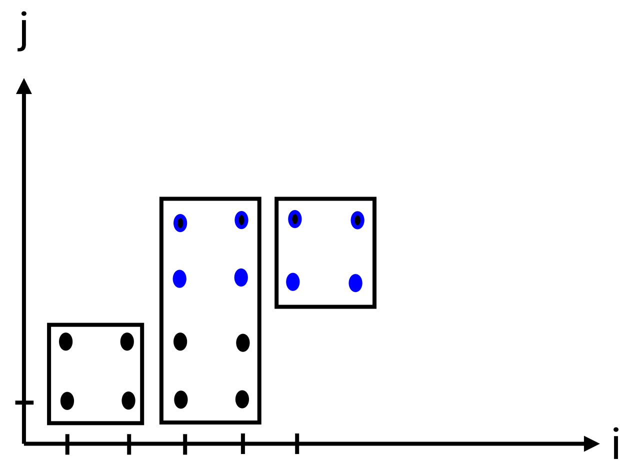
```
for(i=1;i<=4;i++)
        for(j=3;j<=6;j++)
                if(1<=i && i<=4 && 1<=j && j<=2)
                        A(i,j)
                if(3<=j && i<=6 && 3<=j && j<=4)
                        B(i,j)
```



Generating Code

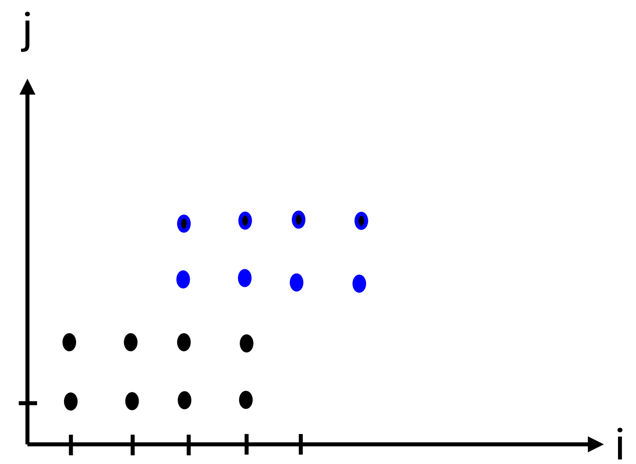
- Turning schedules back into for loops
  - Project onto i-axis
  - Isolate intervals such that group of statements that execute are the same

```
for(i=1;i<=2;i++)
      for(j=1;j<=2;j++)
            A(i,j)
for(i=3;i<=4;i++)
      for(j=1;j<=4;j++)
            if(1<=j && j<=2)
                  A(i,j)
            if(3<=j && j<=4)
                  B(i,j)
for(i=5;i<=6;i++)
      for(j=3;j<=4;j++)
            B(i,j)
```



Generating Code

- Many custom techniques possible
  - Analyzing cost of branches
  - Different decompositions



Summary

- Scalable for few 10s of statements
- Affine statements
- Code generation is complex
- Counting points in the polyhedron is harder
- Great for analysis (and transformation)
    - Polymage labs