Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations

SC14 - New Orleans, Louisiana
November 18th, 2014
Jun Shirako, Louis-Noel Pouchet, Vivek Sarkar
Two Views of Program Representations

**AST (Abstract Syntax Tree) view**
- AST captures all input programs
- Multiple steps modify AST while keeping the semantics

**Polyhedral view**
- Limited to loops whose bounds and accesses are affine expressions
- Single mathematical operation computes optimal solution

dependence S1 to S2:
- $i = i'$
- $k = k'$

S1:
- $0 \leq i \leq n$
- $0 \leq j \leq n$
- $0 \leq k \leq n$

S2:
- $0 \leq i \leq n$
- $0 \leq j \leq n$
- $i \leq k \leq n$
AST-based Loop Transformation Framework

Input: AST-IR

Output: AST-IR

- **Sequence of individual loop transformations on Abstract Syntax Tree**
  - Including: fusion, distribution, permutation, skewing, tiling, unroll-and-jam
  - Each step focuses on specific optimization objective:
    - Parallelism (doall, reduction, pipeline)
    - Temporal and spatial data locality
    - Vectorization efficiency
  - Analysis and cost model customized for each transformation
  - Phase-ordering problem (which comes before/after which)
  - Numerous transformations are complementary to each other
Mathematical Approach to Unified Transformation

- Polyhedral model
  - Algebraic framework for affine program representation and transformation
  - Ability to handle everything in single stage
    - Unified view that captures arbitrary loop structures
    - Generalizes loop transformations as form of affine transform
  - Complexity due to unification/generalization
    - Hard to model cost functions for unified transformations
      - Multiple objectives to be combined in a single cost model
Objective: Minimization of reuse distance
- Better temporal data locality
- Outer parallelism by pushing dependences inside

```c
// Input: sequence of two matmults
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S1: tmp[i][j] += A[i][k] * B[k][j];

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S2: D[i][j] += C[i][k] * tmp[k][j];
```
Objective: Minimization of reuse distance

Better temporal data locality

Outer parallelism by pushing dependences inside
Cost Model Example in Polyhedral Approaches

// Input: sequence of two matmults
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S1: tmp[i][j] += A[i][k] * B[k][j];

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S2: D[i][j] += C[i][k] * tmp[k][j];

// Output: Minimum reuse distance
#pragma omp parallel for private(c2, c3)
for (c1 = 0; c1 < N; c1++)
    for (c2 = 0; c2 < N; c2++)
        for (c3 = 0; c3 < N; c3++)
            S1: tmp[c2][c1] += A[c2][c3] * B[c3][c1];

for (c3 = 0; c3 < N; c3++)
    S2: D[c3][c1] += C[c3][c2] * tmp[c2][c1];

● Objective: Minimization of reuse distance
  ● Better temporal data locality
  ● Outer parallelism by pushing dependences inside
  ● Poor spatial data locality: not modeled in this objective
Mathematical Approach to Unified Transformation

Input: Poly-IR

IR \rightarrow \text{single optimization stage} \rightarrow \text{IR'}

Output: Poly-IR

unified objective

• Challenge : Combining multiple objectives for unified transformations
  • Objectives can conflict, e.g., temporal locality (fuse loop) vs. vectorization (distribute)
Mathematical Approach to Unified Transformation

**Challenge**: Combining multiple objectives for unified transformations
- Objectives can conflict, e.g., temporal locality (fuse loop) vs. vectorization (distribute)

**Our approach** — decouple the optimization problem into two stages with different cost functions:
- **Global** - i.e., inter-loop-nest
  - Good candidate for polyhedral approach
    - Unified view that captures arbitrary loop structures (perfect & imperfect nests)
- **Local** - i.e., per-loop-nest
  - Good candidate for AST-based approach
    - Well-defined sequence of transformations on perfect loop nest
Integrating Polyhedral and AST-based Transformations

- **Poly+AST**: two-stage approach to integration
  - **Stage-1**: Polyhedral transformations
    - Finds optimal loop structures to provide sufficient data locality
      - Restricted form of affine transform
      - Extension of memory cost model for polyhedral model
    - Output: locality-optimized loop nests
  - **Stage-2**: AST-based transformations
    - Input: loop nests and dependences from stage-1
    - Sequence of individual transformations per loop nest (w/ different objectives)
      - Loop skewing (increase tilability)
      - Parallelization (outermost doall / reduction / doacross)
      - Loop tiling (enhance locality and granularity of parallelism)
      - Intra-tile optimization (e.g., register-tiling, if-optimization, ...)


Outline

• Introduction

• Stage-1: Cache-aware polyhedral transformations

• Stage-2: AST-based transformations

• Experimental results vs. stage-of-the-art polyhedral compiler

• Conclusions
Polyhedral Representation of Program

• Iteration domain
  • $\mathcal{D}^S_i$: Set of iteration instances $i = (i_1, i_2, ..., i_n)$ of $S_i$
  • Statement $S_i$ is enclosed in $n$ loops

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      $S_1$: tmp[i][j] += A[i][k] * B[k][j];

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      $S_2$: D[i][j] += C[i][k] * tmp[k][j];

(i, j, k) ∈ $\mathcal{D}^{S_1}$:
0 ≤ i ≤ N-1
0 ≤ j ≤ N-1
0 ≤ k ≤ N-1

(i, j, k) ∈ $\mathcal{D}^{S_2}$:
0 ≤ i ≤ N-1
0 ≤ j ≤ N-1
0 ≤ k ≤ N-1
Polyhedral Representation of Program

- **Iteration domain**
  - $\mathcal{D}_i$: Set of iteration instances $i = (i_1, i_2, ..., i_n)$ of $S_i$
    - Statement $S_i$ is enclosed in $n$ loops

- **Dependence polyhedron**
  - $\mathcal{D}_{i\rightarrow j}$: Captures dependence from $S_i$ to $S_j$
    - $\langle s, t \rangle \in \mathcal{D}_{i\rightarrow j} \iff t \in \mathcal{D}_j$ depends on $s \in \mathcal{D}_i$

```plaintext
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      $S_1$: tmp[i][j] += A[i][k] * B[k][j];

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      $S_2$: D[i][j] += C[i][k] * tmp[k][j];
```

- $(i, j, k) \in \mathcal{D}_{S_1}$: $(i', j', k') \in \mathcal{D}_{S_1 \rightarrow S_2}$:
  - $0 \leq i \leq N-1$
  - $0 \leq j \leq N-1$
  - $0 \leq k \leq N-1$
  - $0 \leq i' \leq N-1$
  - $0 \leq j' \leq N-1$
  - $0 \leq k' \leq N-1$
  - $i = k'$
  - $j = j'$
General Affine Program Transformation

\[ \Theta^S_i(i) = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \ldots & \alpha_{1,d} & c_1 \\ \alpha_{2,1} & \alpha_{2,2} & \ldots & \alpha_{2,d} & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \alpha_{n,1} & \alpha_{n,2} & \ldots & \alpha_{n,d} & c_n \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_d \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} i_1 + \alpha_{1,2} i_2 + \ldots + \alpha_{1,d} i_d + c_1 \\ \alpha_{2,1} i_1 + \alpha_{2,2} i_2 + \ldots + \alpha_{2,d} i_d + c_2 \\ \vdots \\ \alpha_{n,1} i_1 + \alpha_{n,2} i_2 + \ldots + \alpha_{n,d} i_d + c_n \end{pmatrix} \]

\[ i = (i_1, i_2, \ldots, i_d)^T : \text{iteration instances of statement } S_i \]

- **Multi-dimensional affine transform**
  - \( \Theta^S_i \) associates \( i \) with a timestamp - i.e., logical execution date (yy/mm/dd)
  - Can model any composition of loop transformations including:
    - Loop fusion, distribution, permutation, skewing, tiling
- **Legality requirements**
  - For all dependence polyhedra: \( \Theta^S_j(t) > \Theta^S_i(s) , (s, t) \in D^{S_i \rightarrow S_j} \)
Stage-1: Cache-aware Polyhedral Transformations

- **Restricted form of affine transformations**
  - To focus on optimal loop structure to provide sufficient locality
  - Weaker constraints can generate simple (i.e., easy-to-optimize) codes

- **Subsumes the following:**
  - Loop fusion, distribution and code motion
    - Group statements with locality into a loop
  - Loop permutation
    - Optimal loop order to optimize locality
  - Loop reversal and index-set shifting
    - Increase the opportunities of fusion/permutation
  - **No loop skewing (but supported in AST stage)**
    - Changes array access pattern, e.g., \(a[i][j]\) to \(a[i+j][j]\)
    - Can miss spatial locality / affect memory cost analysis
Proposed Restricted Affine Transformation

\[ \Theta_{Si}(i) = \begin{bmatrix} 0 & 0 & \ldots & 0 & \beta_1 \\ \alpha_{1,1} & \alpha_{1,2} & \ldots & \alpha_{1,d} & c_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & \beta_k \\ \alpha_{k,1} & \alpha_{k,2} & \ldots & \alpha_{k,d} & c_k \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & \beta_d \\ \alpha_{d,1} & \alpha_{d,2} & \ldots & \alpha_{d,d} & c_d \\ 0 & 0 & \ldots & 0 & \beta_{d+1} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_d \\ 1 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \alpha_{1,x} \, i_x + c_1 \\ \vdots \\ \beta_k \\ \alpha_{k,y} \, i_y + c_k \\ \vdots \\ \beta_d \\ \alpha_{1,z} \, i_z + c_d \\ \beta_{d+1} \end{bmatrix} \quad \forall k, \sum_{j=1}^{d} |\alpha_{k,j}| = 1 \]

- **Restricted forms**
  - Odd row : constant offset \( \beta_k \)
  - Even row : linear expression of index where coefficient \( \alpha_{k,x} = \pm 1 \)
- **Symbols ⇔ transformations**
  - offset \( \beta_k \) ⇔ fusion / distribution / code motion
  - index \( i_x \) ⇔ permutation
  - coefficient \( \alpha_{k,x} \) ⇔ reversal (apply loop reversal when \( \alpha_{k,x} = -1 \))
  - offset \( c_k \) ⇔ index-set shifting
Cost Model to Guide Polyhedral Transfo.

for $ti = 0, N-1, Ti$
for $tj = 0, M-1, Tj$
for $tk = 0, K-1, Tk$
for $i = ti, ti+Ti-1$
for $j = tj, tj+Tj-1$
for $k = tk, tk+Tk-1$
\[ A[i][j] += B[k][i]; \]

\[ DL(Ti,Tj,Tk) = DL_A(Ti,Tj,Tk) + DL_B(Ti,Tj,Tk) = Ti \times \lceil Tj / L \rceil + Tk \times \lceil Ti / L \rceil \]

\[ \text{mem\_cost}(T_1, T_2, \ldots, T_d) = \text{COST\_LINE} \times DL(T_1, T_2, \ldots, T_d) / (T_1 \times T_2 \times \ldots \times T_d) \]

- **DL (Distinct Line) model**
  - Assumes loop tiling to fit data within cache/TLB
  - Number of Distinct cache Lines accessed within a tile
    - Total cache miss counts per tile

- **Average (per-iteration) memory cost**
  - Defined as [total cache miss penalty per tile] / [tile size]
Profitability Analysis via DL Memory Cost

- **Most profitable loop permutation order**
  - Partial derivative of memory cost w.r.t. $T_k$:
    $$\frac{\partial \text{mem_cost}(T_1, T_2, ..., T_d)}{\partial T_k}$$
  - Reduction rate of memory cost when increasing $T_k$ $\rightarrow$ **Priority of permutation**
    - Loop $k$ with most negative value $\rightarrow$ to be innermost position
    - Best loop order = descending order of $\frac{\partial \text{mem_cost}(T_1, T_2, ..., T_d)}{\partial T_k}$

- **Profitability of loop fusion**
  - Comparing $\text{mem_cost}(T_1, T_2, ..., T_d)$ before/after fusion
    - Memory cost decreased $\rightarrow$ fusion is profitable
      * tentative tile size used; final tile size selected later phase
  - Other criteria, e.g., parallelism, are also considered
Affine Transformation Algorithm

Input: \( S \): set of statements \( S_i \),
\( PoDG \): polyhedral dependence graph,
\( k \): current nest level, or dimension,
\( niter^Si \): \# iterators not yet scheduled in \( \Theta^Si \)

begin
\( PoDG' := \) subset of \( PoDG \) w/o satisfied dependence;
\( SccSet := \) compute SCCs of \( PoDG' \);

/* Intra-SCC transformation (permutation) */
for each \( SCC_a \in SccSet \) do
    compute permutation at level \( k \) and get constraints on reversal (\( \alpha_k,* \)) and shifting (\( c_k \));

/* Inter-SCC transformation (fusion / distribution) */
\( FuseSet := \) compute \( \beta_k \) and get constraints on reversal and shifting;
for each \( Fuse_a \in FuseSet \) do
    solve constraints on reversal and shifting and compute \( \alpha_k,* \) and \( c_k \);
    if \( \exists S_i \in Fuse_a : niter^Si \geq 1 \) then
        recursively process the next level - i.e., \( k+1 \);
end

Output: Dimensions \( k \) ... \( m \) of schedule \( \Theta^Si \)
// Input: sequence of two matmults
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S1: tmp[i][j] += A[i][k] * B[k][j];

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S2: D[i][j] += C[i][k] * tmp[k][j];

<table>
<thead>
<tr>
<th></th>
<th>tmp/D[i][j]</th>
<th>A/C[i][k]</th>
<th>B/tmp[k][j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>N/A</td>
<td>N/A</td>
<td>temporal</td>
</tr>
<tr>
<td>j</td>
<td>spatial</td>
<td>temporal</td>
<td>spatial</td>
</tr>
<tr>
<td>k</td>
<td>temporal</td>
<td>spatial</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Running Example: 2mm

// Input: sequence of two matmults
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S1: tmp[i][j] += A[i][k] * B[k][j];

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            S2: D[i][j] += C[i][k] * tmp[k][j];

// Output: Best permutation order
for (c1 = 0; c1 < N; c1++)  // c1 = i
    for (c2 = 0; c2 < N; c2++)  // c2 = k
        for (c3 = 0; c3 < N; c3++)  // c3 = j
            S1: tmp[c1][c3] += A[c1][c2] * B[c2][c3];

for (c1 = 0; c1 < N; c1++)  // c1 = i
    for (c2 = 0; c2 < N; c2++)  // c2 = k
        for (c3 = 0; c3 < N; c3++)  // c3 = j
            S2: D[c1][c3] += C[c1][c2] * tmp[c2][c3];

<table>
<thead>
<tr>
<th></th>
<th>tmp/D[i][j]</th>
<th>A/C[i][k]</th>
<th>B/tmp[k][j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>N/A</td>
<td>N/A</td>
<td>temporal</td>
</tr>
<tr>
<td>j</td>
<td>spatial</td>
<td>temporal</td>
<td>spatial</td>
</tr>
<tr>
<td>k</td>
<td>temporal</td>
<td>spatial</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- Optimization policy
  - Permute loops as close to the DL best order as possible
  - Fuse loops if legality and profitability criteria are met
Connection between Polyhedral and AST-based Stages

- **Output of polyhedral stage**
  - Locality-optimized loop nests
    - Permutated with legal & profitable loop order
    - Fused statements with locality into a loop
  - Dependence information
    - $\langle s, t \rangle \in P_{e^{Si\rightarrow Sj}}$ : relationship between source and target instances $s$ and $t$
    - Extracted as dependence vector - i.e., $d = t - s$

- **Input of AST-based stage**
  - $loop_k$ : a loop that is nested at level $k \in \{1 \ldots n\}$
  - $\Delta^{loop_k} = \{d^1, d^2, \ldots, d^n\}$:
    - Set of dependences whose source and target statements are within $loop_k$
    - Free from affine constraints in AST-based stage
Stage-2 : AST-based Transformation

- Dependence vectors : base of analysis
  - Legality : loop skewing, loop tiling, register tiling, ...
  - Detection of parallelism

- Sequence of transformations in stage-2
  - Loop skewing
    - In order to increase permutability (i.e., applicability of tiling) and parallelism
  - Coarse-grain parallelization
    - Doall / reduction / doacross parallelism
  - Loop tiling
    - Enhance computation granularity and data locality
  - Intra-tile optimizations
    - Register-tiling (i.e., multi-dimensional unrolling)
Parallelism in Poly+AST Framework

- **Loop permutation order**
  - To optimize spatial and temporal data locality
  - Outermost loop is not always doall
    - **Also leverage other parallelism**: reduction and doacross (pipeline parallelism)

- **Reduction parallelism**

  ```c
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      S[j] += alpha * X[i][j];
  ```

- **Doacross parallelism**

  ```c
  for (i = 1; i < N-1; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0.33 * (C[i-1][j]
                        + C[i][j] + C[i+1][j]);
    }
  }
  ```

OpenMP Extensions:
- "Tile Reduction: The First Step towards Tile aware Parallelization in OpenMP", IWOMP-2009
- "Expressing DOACROSS Loop Dependencies in OpenMP", IWOMP-2013
Parallelism in Poly+AST Framework

• **Loop permutation order**
  - To optimize spatial and temporal data locality
  - Outermost loop is not always doall
    - Also leverage other parallelism: reduction and doacross (pipeline parallelism)

• **Reduction parallelism**
  ```
  #pragma omp for reduction(+: S[0:N-1])
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      S[j] += alpha * X[i][j];
  ```

• **Doacross parallelism**
  ```
  for (i = 1; i < N-1; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0.33 * (C[i-1][j]
                       + C[i][j] + C[i+1][j]);
    }
  }
  ```

OpenMP Extensions:
  - “Tile Reduction: The First Step towards Tile aware Parallelization in OpenMP”, IWOMP-2009
  - “Expressing DOACROSS Loop Dependencies in OpenMP”, IWOMP-2013
Parallelism in Poly+AST Framework

- **Loop permutation order**
  - To optimize spatial and temporal data locality
  - Outermost loop is not always doall
    - Also leverage other parallelism: reduction and doacross (pipeline parallelism)

- **Reduction parallelism**
  ```
  #pragma omp for reduction(+: S[0:N-1])
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      S[j] += alpha * X[i][j];
  ```

- **Doacross parallelism**
  ```
  for (i = 1; i < N-1; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0.33 * (C[i-1][j] + C[i][j] + C[i+1][j]);
    }
  }
  ```

- **Doall-only approach**
  ```
  #pragma omp for
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      S[j] += alpha * X[i][j];
  ```

OpenMP Extensions:
- “*Tile Reduction:The First Step towards Tile aware Parallelization in OpenMP*”, IWOMP-2009
- “*Expressing DOACROSS Loop Dependencies in OpenMP*”, IWOMP-2013
Parallelism in Poly+AST Framework

• Loop permutation order
  • To optimize spatial and temporal data locality
  • Outermost loop is not always doall
    • Also leverage other parallelism: reduction and doacross (pipeline parallelism)

• Reduction parallelism

```c
#pragma omp for reduction(+: S[0:N-1])
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    S[j] += alpha * X[i][j];
```

• Doacross parallelism

```c
#pragma omp for ordered(2)
for (i = 1; i < N-1; i++) {
  for (j = 0; j < N; j++) {
    #pragma omp ordered depend(sink: i-1,j)
    C[i][j] = 0.33 * (C[i-1][j]
                      + C[i][j] + C[i+1][j]);
    #pragma omp ordered depend(src: i,j)
  }
}
```

• Doall-only approach

```c
#pragma omp for
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    S[j] += alpha * X[i][j];
```

OpenMP Extensions:

- “Tile Reduction: The First Step towards Tile aware Parallelization in OpenMP”, IWOMP-2009
- “Expressing DOACROSS Loop Dependencies in OpenMP”, IWOMP-2013
Parallelism in Poly+AST Framework

• **Loop permutation order**
  - To optimize spatial and temporal data locality
  - Outermost loop is not always doall
    - Also leverage other parallelism: reduction and doacross (pipeline parallelism)

• **Reduction parallelism**

```c
#pragma omp for reduction(+: S[0:N-1])
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        S[j] += alpha * X[i][j];
```

• **Doacross parallelism**

```c
#pragma omp for ordered(2)
for (i = 1; i < N-1; i++) {
    for (j = 0; j < N; j++)
        C[i][j] = 0.33 * (C[i-1][j] + C[i][j] + C[i+1][j]);
#pragma omp ordered depend(sink: i-1,j)
    C[i][j] = 0.33 * (C[i-1][j] + C[i][j] + C[i+1][j]);
#pragma omp ordered depend(src: i,j)
}
```

• **Doall-only approach**

```c
#pragma omp for reduction(+: S[0:N-1])
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        S[j] += alpha * X[i][j];
```

OpenMP Extensions:

• “Tile Reduction: The First Step towards Tile aware Parallelization in OpenMP”, IWOMP-2009
• “Expressing DOACROSS Loop Dependencies in OpenMP”, IWOMP-2013
Pipeline Parallelism vs. Wavefront Doall

- **Pipeline parallelism (OpenMP extension)**

```c
#pragma omp parallel for ordered(2)
for (i = 1; i < N-1; i++) {
    for (j = 1; j < N-1; j++) {
        #pragma omp ordered depend(sink: i-1,j)
        depend(sink: i,j-1)
        A[i][j] = A[i-1][j] + a[i][j-1];
        #pragma omp ordered depend(src: i,j)
    }
}
```

- **Wavefront doall with skewing**

```c
#pragma omp parallel
for (i = 2; i <= 2*N-4; i++) {
    #pragma omp for
    for (j = max(1,i-N+2);
         j < min(N-2,i-1); j++) {
        A[i-j][j] = A[i-j-1][j] + a[i-j][j-1];
    }
}
```

---

### Diagrams

- **Pipeline parallelism**

  - Parallel execution with dependencies.
  - **p2p sync** and **seq. region** indicated.

- **Wavefront doall**

  - Skewed execution with barriers.
  - **All-to-all barrier** highlighted.
  - **Red lines** denote parallel execution.
// Input (imperfect nest)
for (t = 0; t < time_steps; t++) {
    for (i = 1; i < n-1; i++)
        S1:   b[i] = 0.33 * (a[i-1] + a[i] + a[i+1]);
            for (i = 1; i < n-1; i++)
        S2:   a[i] = b[i];
    }
}
Another Example: Jacobi-1d stencil

// Input (imperfect nest)
for (t = 0; t < time_steps; t++) {
    for (i = 1; i < n-1; i++)
        S1:  b[i] = 0.33 * (a[i-1] + a[i] + a[i+1]);
    for (i = 1; i < n-1; i++)
        S2:  a[i] = b[i];
}

// Stage-1: polyhedral transformation (perfect nest)
for (c1 = 0; c1 <= time_steps-1; c1++) {
    for (c3 = 1; c3 <= n-1; c3++) {
        S1:  if (c3 <= n-2) b[c3] = 0.33 * (a[c3-1] + a[c3] + a[c3+1]);
        S2:  if (c3 >= 2) a[c3-1] = b[c3-1];
    }
}
Another Example: Jacobi-1d stencil

// Input (imperfect nest)
for (t = 0; t < time_steps; t++) {
    for (i = 1; i < n-1; i++)
        S1:  b[i] = 0.33 * (a[i-1] + a[i] + a[i+1]);
    for (i = 1; i < n-1; i++)
        S2:  a[i] = b[i];
}

// Stage-1: polyhedral transformation (perfect nest)
for (c1 = 0; c1 <= time_steps-1; c1++) {
    for (c3 = 1; c3 <= n-1; c3++) {
        S1:  if (c3 <= n-2) b[c3] = 0.33 * (a[c3-1] + a[c3] + a[c3+1]);
        S2:  if (c3 >= 2) a[c3-1] = b[c3-1];
    }
}

// Stage-2: skewing & parallelization
// - Loop nest is fully permutable
// - Doacross parallelization by OpenMP extensions
#pragma omp parallel for private(c3) ordered(2)
for (c1 = 0; c1 < time_steps; c1++) {
    for (c3 = 2*c1+1; c3 < 2*c1+n; c3++) {
        #pragma omp ordered depend(sink: c1-1,c3) depend (sink: c1,c3-1)
        S1:  if (i <= n-2) b[-2*c1+c3] = 0.33*(a[-2*c1+c3-1]+a[-2*c1+c3]+a[-2*c1+c3+1]);
        S2:  if (i >= 2) a[-2*c1+c3-1] = b[-2*c1+c3-1];
        #pragma omp ordered depend(source: c1,c3)
    }
}
Another Example : Jacobi-1d stencil

// Stage-2: loop tiling
#pragma omp parallel for private(c3,c5,i) ordered(2)
for (c1 = ...) {
    for (c3 = ...) {
        #pragma omp ordered depend(sink: c1-1,c3) depend(sink: c1,c3-1)

        for (c5 = ...) {
            for (c7 = ...) {
                S1: b[-2*c5+c7] = 0.33 * (a[-2*c5+c7-1] + a[-2*c5+c7] + a[-2*c5+c7+1]);
                S2: a[-2*c5+c7-1] = b[-2*c5+c7-1];
            }
            if (...) A[n-2] = B[n-2];
        }
    }
    #pragma omp ordered depend(source: c1,c3)
} }
// Stage-2: loop tiling
#pragma omp parallel for private(c3,c5,i) ordered(2)
for (c1 = ...) {
    for (c3 = ...) {
        #pragma omp ordered depend(sink: c1-1,c3) depend(sink: c1,c3-1)
        ...
            for (c5 = ...) {
                if (...)
                for (c7 = ...) {
                    S1:  b[-2*c5+c7] = 0.33 * (a[-2*c5+c7-1] + a[-2*c5+c7] + a[-2*c5+c7+1]);
                    S2:  a[-2*c5+c7-1] = b[-2*c5+c7-1];
                        }
                if (...)
            }

        #pragma omp ordered depend(source: c1,c3)
    }
}

// Stage-2: register tiling (innermost by factor = 2)
...
for (c7 = ...; c7 <= (......)-1; c7+=2) {
    S1:  b[-2*c5+c7] = 0.33 * (a[-2*c5+c7-1]+a[-2*c5+c7]+a[-2*c5+c7+1]);
    S2:  a[-2*c5+c7-1] = b[-2*c5+c7-1];
    S1': b[-2*c5+c7+1] = 0.33 * (a[-2*c5+c7+1-1]+a[-2*c5+c1]+a[-2*c5+c7+1+1]);
    S2': a[-2*c5+c7+1-1] = b[-2*c5+c7+1-1];
} ...
Experimental Setting

- **Platforms**
  - Two quad-core 2.8GHz Intel Core i7 (Nehalem) with Intel C compiler 12.0
  - Four eight-core 3.86GHz IBM Power7 with IBM XLC compiler 11.1

- **Benchmarks**
  - PolyBench-C 3.2 (22 benchmarks, standard/large dataset)

- **Comparisons**
  - PoCC: research polyhedral compiler [http://www.cs.ucla.edu/~pouchet/software/pocc]
    - PLuTo heuristic for parallelism, locality, tiling and intra-tile optimizations
    - Doall parallelism (convert doacross into wavefront doall)
  - PoCC-iterative: Iterative compilation approach [Pouchet-SC’10]
    - PoCC + empirical search for outermost fusion/distribution
  - Poly+AST: proposed integration approach
    - Doall / doacross / reduction parallelism

- **Additional results in paper, e.g., ICC and XLC**
GFLOP/s on Nehalem (doall dominant)

- PoCC-iterative: empirical search for fusion/distribution
- Poly+AST (polyhedral stage): DL model for fusion/dist. and permutation

PoCC ≤ PoCC-iterative ≤ Poly+AST
GFLOP/s on Nehalem (doacross-parallel dominant)

- PoCC = PoCC-iterative ≤ Poly+AST
- adi / cholesky / fdtd-2d: loop structures (e.g., fusion, perm., index-shifting)
- jacobi-2d: DOACROSS parallelization vs. wavefront doall by skewing
GFLOP/s on Nehalem (with reduction parallelism)

- PoCC ≤ PoCC-iterative < Poly+AST
- Reduction support to increase flexibility of loop permutation
- Loop order w/ better locality while keeping outermost parallelism
// Input: SYMM (simplified)
for (i = 0; i < NI; i++) {
    for (j = 0; j < NJ; j++) {
        for (k = 0; k < j - 1; k++) {
            S1: C[k][j] += alpha * A[k][i] * B[i][j];
            S2: acc[i][j] += B[k][j] * A[k][i];
        }
        S3: C[i][j] = beta * C[i][j] + alpha * A[i][i]* B[i][j] + alpha * acc[i][j];
    }
}
// PoCC optimized (omitting tiling and intra-tile optimizations)
#pragma omp parallel for private(c2, c3)
for (c1 = 2; c1 <= NJ-1; c1++) {
    for (c2 = 0; c2 <= NI-1; c2++) {
        for (c3 = 0; c3 <= c1+NI-1; c3++) {
            S1: if (c3 <= c1-2) acc[c2][c1] += B[c3][c1] * A[c3][c2];
            S2: if (c2 <= c1-2 && c3 >= c1) C[c2][c1] += alpha * A[-c1+c3][c2][c1];
            S3: if (c3 == c1+c2) C[c2][c1] = beta * C[c2][c1] + alpha * A[c2][c2] * B[c2][c1] ...
        }
    }
}

doall accessing inner array dimensions; poor spatial locality
Transformed Codes by PoCC and Poly+AST

// PoCC optimized (omitting tiling and intra-tile optimizations)
#pragma omp parallel for private(c2, c3)
for (c1 = 2; c1 <= NJ-1; c1++) {
    for (c2 = 0; c2 <= NI-1; c2++) {
        for (c3 = 0; c3 <= c1+NI-1; c3++) {
            S1: if (c3 <= c1-2) acc[c2][c1] += B[c3][c1] * A[c3][c2];
            S2: if (c2 <= c1-2 && c3 >= c1) C[c2][c1] += alpha * A[c2][-c1+c3] * B[-c1+c3][c1];
            S3: if (c3 == c1+c2) C[c2][c1] = beta * C[c2][c1] + alpha * A[c2][c2] * B[c2][c1] ...
        }
    }
}
doall accessing inner array dimensions; poor spatial locality

// Poly+AST optimized (omitting tiling and intra-tile optimizations)
#pragma omp parallel for private(c3, c5) reduction(+: acc[0:NI-1][2:NJ-1])
for (c1 = 0; c1 <= NJ-3; c1++) {
    for (c3 = 0; c3 <= NI-1; c3++) {
        for (c5 = c1 + 2; c5 <= NJ-1; c5++) {
            S1: acc[c3][c5] += B[c1][c5] * A[c1][c3];
        }
    }
#pragma omp parallel for private(c3, c5)
for (c1 = 0; c1 <= MAX(NI-1, NJ-3); c1++) {
    for (c3 = 0; c3 <= NI-1; c3++) {
        for (c5 = 0; c5 <= NJ-1; c5++) {
            S2: if (c5 >= c1+2) C[c1][c5] += alpha * A[c1][c3] * B[c3][c5];
            S3: if (c3 == c1) C[c1][c5] = beta * C[c1][c5] + alpha * A[c1][c1] * B[c1][c5] ...
        }
    }
} }
reduction / doall accessing outer array dimensions; better spatial locality
GFLOP/s on Power7 (doall dominant)

- PoCC = PoCC-iterative \leq \text{Poly+AST}
- Good selection of loop structures (e.g., fusion/distribution and permutation)
GFLOP/s on Power7 (doacross-parallel dominant)

- PoCC = PoCC-iterative ≤ Poly+AST
- Efficiency of DOACROSS has more impact (32-core Power7 vs. 8-core Nehalem)
GFLOP/s on Power7 (with reduction parallelism)

- Reduction reduces performance (correl, covar and symm)
- Sequential aggregation for final results is scalability bottleneck
- Future work: parallel aggregation
**Take-home Message**

- **AST-based transformations**
  - Sequence of individual loop transformations
  - Difficulty in composing the optimal sequence (i.e., phase-ordering)
- **Polyhedral model**
  - Unification & generalization of loop transformations
  - Difficulty in modeling cost functions for whole unified transformations
- **Integration of both**
  - Decoupling the optimization problem into two stages
    - Polyhedral model as first stage, AST-based as second stage
  - Simpler & customized cost modeling within stage
  - Each stage leverage its strengths
  - Geometric mean speedup vs. PoCC (polyhedral optimizer)
    - 1.62x on 8-core Nehalem / 1.49x on 32-core Power7
We are grateful to P. Sadayappan at The Ohio State University and J. Ramanujam at Louisiana State University for their feedback and discussions on the polyhedral model.