Compiler-Based Autotuning Technology

Lecture 3: A Closer Look at Polyhedral Compiler Technology

Mary Hall
July, 2011

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.
Definition:
- Represent iteration spaces of loop nests as sets of integer-valued points in regions of spaces
- A set $S$ is a polyhedron if it can be represented by a system of inequalities $Ax \leq b$

Advantages:
- Mathematical representation provides elegant and robust representation for manipulation and code generation
- Suitable for loop nest computations, where subscripts and loop bounds are affine

Systems dating back to early 1990s, but renewed interest and production implementations in recent years
- Graphite (gcc), Polly (LLVM), R-Stream (Reservoir), Omega, CLooG, PLUTO, ISL, piplib, PPL, LooPo,...
Outline for Today’s Lecture

1. Abstractions
   a. Dependence graph
   b. Iteration space representation
   c. Code transformations rewrite iteration spaces
   d. Scanning polyhedra for code generation

2. More transformations: tiling, unroll-and-jam

3. Advanced concepts for imperfect loop nests
   a. Sequencing statements
   b. Aligning iteration spaces
   c. Code generation for imperfect loop nests

4. Extended example: LU without pivoting
1. Guide to Abstractions

a. build dependence graph

b. represent iteration spaces

code: ........................................

statements + iteration spaces

c. Loop transformations
   permute
   tile
   unroll-and-jam
   ...

statements + transformed iteration spaces

d. code generation

transformed code

code: ........................................
1. Guide to Implementation

CHiLL
Driver and Transformation Algorithms

Omega+
Solves Constraints and Represents Integer Sets

Codegen+
Generates Loop Code by Scanning Polytopes

Compiler Internal Representation, Abstract Syntax Tree
1. Example: Matrix-Vector Multiply

for (i=0; i<100; i++)
    for (j=0; j<50; j++)
        a[i] = a[i] + c[j][i]*b[j];
1a. Guide to Abstractions: Dependence Graph

- **Original code**
- **Transformation script**
- **Build dependence graph**
- **Represent iteration spaces**
- **Loop transformations** (permute, tile, unroll-and-jam, ...)
- **Code generation**

**Statements + iteration spaces**

**Statements + transformed iteration spaces**

**Transformed code**
1a. Data Dependence

- **Definition:**
  A *data dependence* is an ordering on a pair of memory operations that must be preserved to maintain correctness.

  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- Two important uses of data dependence information (among others):
  - **Parallelization:** no data dependence between two computations → parallel execution safe
  - **Locality optimization:** absence of data dependences & presence of reuse → reorder memory accesses for better data locality
1a. Data Dependence of Scalar Variables

**True (flow) dependence**
\[
\begin{align*}
a &= a
\end{align*}
\]

**Anti-dependence**
\[
\begin{align*}
a &= a
\end{align*}
\]

**Output dependence**
\[
\begin{align*}
a &= a
\end{align*}
\]

**Input dependence (for locality)**
\[
\begin{align*}
a &= a
\end{align*}
\]

**Definition:**
Data dependence exists from a reference instance I to I’ iff
- either i or i’ is a write operation
- I and I’ refer to the same variable
- I executes before I’
1a. Fundamental Theorem of Dependence

- Theorem 2.2 from Allen/Kennedy:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Result: Use data dependence analysis to determine whether dependences are preserved by transformations, including parallelization.

1a. Data Dependence of Array Variables
Equivalence to Integer Programming

- Determine if $F(I) = G(I')$, where $I$ and $I'$ are iteration vectors, with constraints $I, I' \geq L$, $U \geq I, I'$

- Example:

  ```
  for (i=1; i<=100; i++)
      A[i] = A[i-1];
  ```

- Inequalities:

  ```
  1 \leq iw \leq 100, \quad ir = iw - 1, \quad ir \leq 100
  ```

  `integer vector I`, `AI \leq b`

- Integer Programming is NP-complete

  - $O$(size of the coefficients)
  - $O(n^n)$
1a. Calculating Data Dependences using Omega+ Calculator

- **Example:**
  ```c
  for (i=2; i<=100; i++)
      A[i] = A[i-1];
  ```

- Define relation $iw = i$, and $iw = ir-1$ in the iteration space $2 \leq i \leq 100$.

  $$
  R := \{[iw] \rightarrow [ir] : \\
  2 \leq iw, ir \leq 100 \quad /* \text{iteration space} */ \\
  \quad \&\& iw < ir \quad /* \text{looking for loop-carried true dep} */ \\
  \quad \&\& iw = ir-1\}; \\
  $$

  $$
  R := \{[iw] \rightarrow [ir] : 2 \leq iw, ir \leq 100 \&\& iw < ir \&\& iw = ir - 1\}; \\
  $$

- **Result:** $\{[iw] \rightarrow [iw+1] : 2 \leq iw \leq 99\}$
1a. Dependences in Matrix-Vector Multiply

```
for (i=0; i<100; i++)
  for (j=0; j<50; j++)
    a[i] = a[i] + c[j][i]*b[j];
```
1a. Dependences in Matrix-Vector Multiply

\[
\begin{align*}
\text{for (i=0; i<100; i++)} \\
\text{for (j=0; j<50; j++)} \\
a[i] &= a[i] + c[j][i]*b[j];
\end{align*}
\]

- $b$ and $c$ are read only: *no dependence*
- Each $I=[i,j]$ iteration accesses the same $a[i]$ for all 50 values of $j$: *dependence "carried" by $j$ loop*
1a. How Dependences are Used in CHiLL

- Dependence graph analyzed to determine safety of code transformations and determine correctness
- After each transformation, the dependence graph is updated to maintain consistency
- An annotation allows the user to indicate that certain dependences can be ignored by the system (related to $IVDEP$ in vectorizing compilers)

In remainder of course, we will not discuss dependences, but their careful handling is essential to guarantee correctness
1b. Guide to Abstractions: Iteration Spaces

code: 
........ 
........

original code

b. represent iteration spaces

a. build dependence graph

statements + iteration spaces

transformation script

c. Loop transformations
permute
tile
unroll-and-jam
...

statements + transformed iteration spaces

d. code generation

code: 
........ 
........

transformed code
1b. Represent Loop Nest Iteration Space

for (i=0; i<100; i++)
    for (j=0; j<50; j++)
        a[i] = a[i] + c[j][i]*b[j];

Iteration space defined by:

I := {[l_1,...,l_n] : LB_1 <= l_1 < UB_1 && ... LB_n <= l_n < UB_n};

In this case:

I1 := {[i,j] : 0 <= i < 100 && 0 <= j < 50};
1c. Guide to Abstractions: Transformations

- **a. build dependence graph**
  - statements + iteration spaces
- **b. represent iteration spaces**
  - transformed iteration spaces
- **c. Loop transformations**
  - permute
  - tile
  - unroll-and-jam
- **d. code generation**

Original code

Transformation script

Transformed code
Initial iteration space:
\[ I_1 := \{[i,j] : 0 \leq i < 100 \land 0 \leq j < 50 \}; \]

Permutation:
\[ P := \{[i,j] \rightarrow [j,i] \}; \]

1c. Transformations Manipulate Iteration Space

\[
\text{for } (i=0; \ i<100; \ i++) \\
\text{for } (j=0; \ j<50; \ j++) \\
a[i] = a[i] + c[j][i]*b[j];
\]
1d. Guide to Abstractions: Code Generation

- **Original code**
- **Build dependence graph**
  - Statements + iteration spaces
- **Represent iteration spaces**
  - Loop transformations
    - Permute
    - Tile
    - Unroll-and-jam
    ... 
  - Statements + transformed iteration spaces
- **Code generation**
  - Transformed code

ACACES 2011, L3: Polyhedral Compiler Technology
Initial iteration space:
\[ I_1 := \{(i,j) : 0 \leq i < 100 \text{ and } 0 \leq j < 50\}; \]

Permutation:
\[ P := \{(i,j) \rightarrow (j,i)\}; \]

Generate code:
\[ \text{codegen } P : I_1; \]

Output of codegen \( I_1 \)
\[
\begin{align*}
\text{for}(t1 = 0; t1 \leq 99; t1++) & \{ \\
& \text{for}(t2 = 0; t2 \leq 49; t2++) \{ \\
& \quad s1(t1,t2); \\
& \}
\}
\end{align*}
\]

Output of codegen \( P : I_1 \)
\[
\begin{align*}
\text{for}(t1 = 0; t1 \leq 49; t1++) & \{ \\
& \text{for}(t2 = 0; t2 \leq 99; t2++) \{ \\
& \quad s1(t2,t1); \\
& \}
\}
\end{align*}
\]

\[ \text{for } (i=0; i<100; i++) \] 
\[ \text{for } (j=0; j<50; j++) \]
\[ a[i] = a[i] + c[j][i]*b[j]; \]
2. More Transformations: Tiling

Initial iteration space:
\[ I_1 := \{(i,j) : 0 \leq i < 100 \land 0 \leq j < 50\}; \]

Tiling (i loop, tile size = 4):
\[ T := \{(i,j) \mapsto (ii,i,j) : \exists a : ii = 4a \land a \geq 0 \land ii \leq i < ii + 4\}; \]

Generate code:
\[
\text{codegen } T: I_1;
\]

Output of codegen \( I_1 \)
\[
\begin{array}{l}
\text{for } (t1 = 0; t1 \leq 99; t1++) \\
\text{for } (t2 = t1; t2 \leq t1+3; t2++) \\
\text{for } (t3 = 0; t3 \leq 49; t3++) \\
\text{s1}(t1,t2); \\
\end{array}
\]

Output of codegen \( T: I_1 \)
\[
\begin{array}{l}
\text{for } (t1 = 0; t1 \leq 96; t1 += 4) \\
\text{for } (t2 = t1; t2 \leq t1+3; t2++) \\
\text{for } (t3 = 0; t3 \leq 49; t3++) \\
\text{s1}(t2,t3); \\
\end{array}
\]

\[
\text{for } (i=0; i<100; i++) \\
\text{for } (j=0; j<50; j++) \\
a[i] = a[i] + c[j][i]*b[j];
\]

for (i=0; i<100; i++)
  for (j=0; j<=i; j++)
    c[i][j] += val;

Initial iteration space:
  I1 := \{[i,j] : 0 \leq i < 100 \&\& 0 \leq j \leq i\};

Unrolling (i loop, unroll factor = 2):
  s0: c[i][j] += val; s1: c[i+1][j] += val;
  r0:=[i,j]: exists (a: i=2a \&\& 0\leq i<100 \&\& 0\leq j\leq i); 
  r1:=[i,j]: exists (a: i=2a \&\& 0\leq i<100 \&\& 0\leq j\leq i+1); 

Generate code:
  codegen r0, r1;

Output of codegen r0, r1:
  for(t1 = 0; t1 <= 98; t1 += 2) {
    for(t2 = 0; t2 <= t1; t2++) {
      s1(t1,t2);
      s2(t1,t2);
    }
    s2(t1,t1+1);
  }
Suppose each vector element is initialized to 0.

How do we represent imperfect iteration spaces?

```c
for (i=0; i<100; i++)
    s0: a[i] = 0;
    for (j=0; j<50; j++)
        s1: a[i] = a[i] + c[j][i]*b[j];
```
3a. Advanced Concepts: Sequencing in Imperfect Loop Nests

We add an auxiliary loop to sequence subloops in an imperfect nest.

\[
\begin{align*}
\text{for } (i=0; i<100; i++) \\
\text{s0: } a[i] &= 0; \\
&\quad \text{for } (j=0; j<50; j++) \\
\text{s1: } a[i] &= a[i] + c[j][i]*b[j];
\end{align*}
\]

\[
\begin{align*}
l(s0) &= \{[0,i,0,j] : 0 \leq i < 100 \land j = 0\}; \\
l(s1) &= \{[0,i,1,j] : 0 \leq i < 100 \land 0 \leq j < 50\};
\end{align*}
\]
3b. Advanced Concepts: Aligning Imperfect Loop Nests to a Common Iteration Space

Alignment example:

```
for (i=0; i<n; i++) {
    s0: sum[i] = 0;
        for (j=0; j<i-1; j++)
    s1: sum[i] = sum[i] + a[j][i] + b[j];
    s2: b[i] = b[i] – sum[i];
}
```

\[ I(s0) := \{[0,i,0,j] : 0 <= i < n \land j = 0\} \]

\[ I(s1) := \{[0,i,1,j] : 0 <= i < 100 \land 0 <= j < i-1\} \]

\[ I(s1) := \{[0,i,2,j] : 0 <= i < 100 \land j = i-2\} \]

Alternative alignment for s2 (j=n-2) leads to less efficient code.
3b. Advanced Concepts: Code Generation of Imperfect Loop Nests

**Iteration spaces:**
- \( r_1 : \{ [0, i, 0, j] : 0 \leq i < 100 \land j = 0 \} \)
- \( r_2 : \{ [0, i, 1, j] : 0 \leq i < 100 \land 1 \leq j < 50 \} \)
- \( r_3 : \{ [0, i, 1, j] : 0 \leq i, j < 50 \} \)

**Output of codegen \( r_1, r_2, r_3; \)**

```c
for (t2 = 0; t2 <= 99; t2++) {
    s1(0, t2, 0, 0);
    if (t2 <= 49) {
        for (t4 = 0; t4 <= 49; t4++) {
            s2(0, t2, 1, t4);
            s3(0, t2, 1, t4);
        }
    }
    if (t2 >= 50) {
        for (t4 = 0; t4 <= 49; t4++) {
            s2(0, t2, 1, t4);
        }
    }
}
```

- **Code generation optimizes the combining of iteration spaces to derive efficient results in the presence of imperfect loop nests**
4. LU Decomposition: Abstractions

- **a.** Build dependence graph
- **b.** Represent iteration spaces
  - **c.** Loop transformations
    - permute
    - tile
    - unroll-and-jam
    - ...
  - statements + iteration spaces
  - statements + transformed iteration spaces
- **d.** Code generation
  - transformed code

Original code:
```
......
......
```

Transformation script:
```
```

Transferred code:
```
......
......
```
4. LU Decomposition: Abstractions

**Code generation**

```
I(s_1): \{[k,i,j] \mid 1 \leq k \leq N-1 \& \& k+1 \leq i \leq N \& \& j=k+1\}
I(s_2): \{[k,i,j] \mid 1 \leq k \leq N-1 \& \& k+1 \leq i,j \leq N\}
```

**Extract representation**

```
DO K=1,N-1
  DO I=K+1,N
    s1 \hspace{1cm} A(I,K)=A(I,K)/A(K,K)
    DO I=K+1,N
      DO J=K+1,N
        s2 \hspace{1cm} A(I,J)=A(I,J)-A(I,K)*A(K,J)
```

**Transformation script**

```
t1 := \{ [k,i,j] \rightarrow [ 0, jj, 0, kk, 0, j, 0, i, 0, k, 0 ] : jj=2+16\beta \& \& kk = 1+128\alpha \& \& j-15, 2 \leq jj \leq j \& \& kk-127, 1 \leq kk \leq k \}
t2 := \{ [k,i,j] \rightarrow [ 0, jj, 0, kk, 0, j, 0, i, 1, k, 0 ] : jj=2+16\beta \& \& kk = 1+128\alpha \& \& j-15, 2 \leq jj \leq j \& \& kk-127, 1 \leq kk \leq k \}
```

**Dependence analysis**

```
flow(0,0,0)
flow(0,0,+) 
flow(0,+,0) 
flow(0,+,+)
flow(+,0,0)
flow(+,0,+) 
flow(+,+,0)
flow(+,+,+)
anti(0,0,1)
anti(0,+,1)
output(0,0,1)
output(0,+,1)
output(+,0,1)
output(+,+,1)
```

**Dependence graph**

```
S1
S2
```

**Transformed code**

```
DO K=1,N-1
  DO I=K+1,N
    s1      A(I,K)=A(I,K)/A(K,K)
    DO I=K+1,N
      DO J=K+1,N
        s2        A(I,J)=A(I,J)-A(I,K)*A(K,J)
```
4. CHiLL Transformation Script for LU

\[ \text{DO } K = 1, N-1 \]
\[ \text{DO } I = K+1, N \]
\[ s1 \quad A(I,K) = A(I,K) / A(K,K) \]
\[ \text{DO } I = K+1, N \]
\[ \text{DO } J = K+1, N \]
\[ s2 \quad A(I,J) = A(I,J) - A(I,K) * A(K,J) \]

Separate perfect and imperfect loop nests

Separate non-overlapping read and write accesses

CHiLL Script Source: Chun Chen
4. Automatically-Generated LU Code

```plaintext
REAL*8 P1(32,32), P2(32,64), P3(32,32), P4(32,64)
OVER1=0
OVER2=0
DO T2=2,N,64
  IF (66<=T2)
    DO T4=2,T2-32,32
      DO T6=1,T4-1,32
        DO T8=T6,MIN(T4-1,T6+31)
          DO T10=T4,MIN(T2-2,T4+31)
            P1(T8-T6+1,T10-T4+1)=A(T10,T8)
          END DO
        END DO
        DO T8=T2,MIN(T2+63,N)
          DO T10=T6,MIN(T6+31,T4-1)
            P2(T10-T6+1,T8-T2+1)=A(T10,T8)
          END DO
        END DO
        OVER1=MOD(-1+N,4)
      END DO
    END IF
    DO T10=T2,MIN(N-OVER1,T2+60),4
      DO T12=T6,MIN(T6+31,T4-1)
        A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
        A(T8,T10+1)=A(T8,T10+1)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+1-T2+1)
        A(T8,T10+2)=A(T8,T10+2)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+2-T2+1)
        A(T8,T10+3)=A(T8,T10+3)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+3-T2+1)
      END DO
    END DO
  END IF
  DO T10=MAX(N-OVER1+1,T2),MIN(T2+63,N)
    DO T12=T6,MIN(T4-1,T6+31)
      A(T8,T10)=A(T8,T10)-A(T6,T10)*A(T10,T8)
    END DO
  END DO
END DO
```

ACACES 2011, L3: Polyhedral Compiler Technology
4. Automatically-Generated LU Code

IF (66<=T2)
DO T4=1,T2-33,32
DO T6=T2-1,N,32
   DO T8=T4,T4+31
      DO T10=T6,MIN(N,T6+31)
         P3(T8-T4+1,T10-T6+1)=A(T10,T8)
      DO T8=T2,MIN(T2+63,N)
         DO T10=T4,T4+31
         P4(T10-T4+1,T8-T2+1)=A(T10,T8)
      DO T8=T6,MIN(T6+31,N)
   OVER2=MOD(-1+N,4)
      DO T10=T2,MIN(N-OVER2,T2+60),4
         DO T12=T4,T4+31
            A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
            A(T8,T10+1)=A(T8,T10+1)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+1-T2+1)
            A(T8,T10+2)=A(T8,T10+2)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+2-T2+1)
            A(T8,T10+3)=A(T8,T10+3)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+3-T2+1)
      DO T10=MAX(T2,N-OVER2+1),MIN(T2+63,N)
         DO T12=T4,T4+31
            A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
      DO T4=T2-1,MIN(N-1,T2+62)
      DO T8=T4+1,N
         A(T8,T4)=A(T8,T4)/A(T4,T4)
      DO T6=T4+1,MIN(T2+63,N)
      DO T8=T4+1,N
         A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)

ACACES 2011, L3: Polyhedral Compiler Technology
Summary of Lecture

• Polyhedral compiler frameworks becoming more common
  - Mathematical manipulation of iteration spaces for transformations and code generation
  - Mostly applicable to affine domain

• Key concepts/abstractions
  - Dependence graph
  - Iteration spaces
  - Transformations rewrite iteration spaces
  - Code generation scans resulting iteration spaces to convert back to loops

• CHiLL-specific concepts
  - Auxiliary loops and alignment represent imperfect loop nests
  - Transformation and code generation algorithms manipulate this expanded iteration space
Other polyhedral and related compiler frameworks.


