Loop Transformations

- Motivation
- Loop level transformations catalogus
  - Loop merging
  - Loop interchange
  - Loop unrolling
  - Unroll-and-Jam
  - Loop tiling
- Loop Transformation Theory and Dependence Analysis

- Change the order in which the iteration space is traversed.
- Can expose parallelism, increase available ILP, or improve memory behavior.
- Dependence testing is required to check validity of transformation.
Why loop transformations: Examples

Example 1: in-place mapping

```
for (j=0; j<M; j++)
  for (i=0; i<N; i++)
    A[i][j] = f(A[i][j-1]);

for (i=0; i<N; i++)
  OUT[i] = A[i][M-1];
```

Example 2: memory allocation

```
for (i=0; i<N; i++)
  B[i] = f(A[i]);
for (i=0; i<N; i++)
  C[i] = g(B[i]);
```

2 background ports

```
for (i=0; i<N; i++)
  B[i] = f(A[i]);
for (j=0; j<M; j++)
  A[i][j] = f(A[i][j-1]);

for (i=0; i=N; i++)
  OUT[i] = A[i][M-1];
```

1 backgr. + 1 foregr. ports
Loop transformation catalogue

- merge
  - improve locality
- bump
- extend/reduce
- body split
- reverse
  - improve regularity
- interchange
  - improve locality/regularity
- skew
- tiling
- index split
- unroll
- unroll-and-jam

Loop Merge

\[
\text{for } I_a = \exp_1 \text{ to } \exp_2 \\
A(I_a) \\
\text{for } I_b = \exp_1 \text{ to } \exp_2 \\
B(I_b)
\]

\[
\Rightarrow \\
\text{for } I = \exp_1 \text{ to } \exp_2 \\
A(I) \\
B(I)
\]

- Improve locality
- Reduce loop overhead
Loop Merge:
Example of locality improvement

```
for (i=0; i<N; i++)
    B[i] = f(A[i]);
for (j=0; j<N; j++)
    C[j] = f(B[j], A[j]);
```

Consumptions of second loop closer to productions and consumptions of first loop

- Is this always the case?

Loop Merge:
Satisfy dependencies

- Data dependencies from first to second loop can block Loop Merge
- Dependency is allowed if \( \forall I: \text{cons}(I) \leq \text{prod}(I) \) in loop 2
- Enablers: Bump, Reverse, Skew

```
for (i=0; i<N; i++)
    B[i] = f(A[i]);
for (i=0; i<N; i++)
    C[i] = f(B[i], A[i]);
```

```
for (i=0; i<N; i++)
    B[i] = f(A[i]);
for (j=0; j<N; j++)
    C[j] = f(B[j], A[j]);
```
Loop Bump

for \( I = \exp_1 \) to \( \exp_2 \)
\[ A(I) \]

\[ \Rightarrow \]

for \( I = \exp_1 + N \) to \( \exp_2 + N \)
\[ A(I-N) \]

Loop Bump: Example as enabler

for \( i=2 \); \( i < N; \) \( i++ \)
\[ B[i] = f(A[i]); \]
for \( i=0; \) \( i < N-2; \) \( i++ \)
\[ C[i] = g(B[i+2]); \]

\( i+2 > i \Rightarrow \) merging not possible

for \( i=2; \) \( i < N; \) \( i++ \)
\[ B[i] = f(A[i]); \]
for \( i=2; \) \( i < N; \) \( i++ \)
\[ C[i-2] = g(B[i+2-2]); \]

\( i+2-2 = i \Rightarrow \) merging possible

Loop Bump

\[ \Rightarrow \]

Loop Merge

\[ \Rightarrow \]
Loop Extend

for \( I = \text{exp}_1 \) to \( \text{exp}_2 \)
\[ A(I) \]
\[
\begin{align*}
\text{exp}_3 & \leq \text{exp}_1 \\
\text{exp}_4 & \geq \text{exp}_2
\end{align*}
\]

\[ \implies \]

for \( I = \text{exp}_3 \) to \( \text{exp}_4 \)
\[
\text{if } I \geq \text{exp}_1 \text{ and } I < \text{exp}_2 \\
A(I)
\]

Loop Extend:
Example as enabler

for \( (i=0; \ i<N; \ i++) \)
\[ B[i] = f(A[i]) \]
for \( (i=2; \ i<N+2; \ i++) \)
\[ C[i-2] = g(B[i]) \]

\[ \implies \]

Loop Extend

\[ \implies \]

for \( (i=0; \ i<N+2; \ i++) \)
\[
\begin{align*}
&\text{if}(i<N) \\
&B[i] = f(A[i]) \\
&\text{if}(i>=2) \\
&C[i-2] = g(B[i])
\end{align*}
\]

\[ \implies \]

Loop Merge

\[ \implies \]

for \( (i=0; \ i<N+2; \ i++) \)
\[
\begin{align*}
&\text{if}(i<N) \\
&B[i] = f(A[i]) \\
&\text{if}(i>=2) \\
&C[i-2] = g(B[i])
\end{align*}
\]
**Loop Reduce**

For I = \( \exp_1 \) to \( \exp_2 \)

if \( I \geq \exp_3 \) and \( I < \exp_4 \)

A(I)

\[ \Rightarrow \]

for I = \( \max(\exp_1, \exp_3) \) to \( \min(\exp_2, \exp_4) \)

A(I)

**Loop Body Split**

For I = \( \exp_1 \) to \( \exp_2 \)

A(I)

B(I)

\( A(I) \) must be single-assignment

\[ \Rightarrow \]

for \( I_a = \exp_1 \) to \( \exp_2 \)

A(I_a)

for \( I_b = \exp_1 \) to \( \exp_2 \)

B(I_b)
Loop Body Split: Example as enabler

```c
for (i=1; i<N; i++)
    A[i] = f(A[i-1]);
    B[i] = g(in[i]);
for (j=1; j<N; j++)
    C[i] = h(B[i],A[N]);
```

Loop Body Split

```c
for (i=1; i<N; i++)
    A[i] = f(A[i-1]);
    B[i] = g(in[i]);
    C[i] = h(B[i],A[N]);
```

Loop Merge

Loop Reverse

```c
for l = exp_1 to exp_2
    A(l)
```

OR

```c
for l = exp_2 downto exp_1
    A(l)
```

```c
for l = exp_1 to exp_2
    A(exp_2-(l-exp_1))
```
Loop Reverse: Satisfy dependencies

No loop-carried dependencies allowed!

Enabler: data-flow transformations (associative)

Loop Reverse: Example as enabler

N–i > i ⇒ merging not possible

Loop Reverse ⇒

N-(N-i) = i ⇒ merging possible

Loop Merge ⇒
Loop Interchange

```
for l₁ = exp₁ to exp₂
  for l₂ = exp₃ to exp₄
    A(l₁, l₂)
```

⇒

```
for l₂ = exp₃ to exp₄
  for l₁ = exp₁ to exp₂
    A(l₁, l₂)
```

Loop Interchange

```
for(i=0; i<W; i++)
  for(j=0; j<H; j++)
    A[i][j] = ...
```

```
for(j=0; j<H; j++)
  for(i=0; i<W; i++)
    A[i][j] = ...
```

Loop Interchange
Loop Interchange

- Validity: dependence direction vectors.
- Mostly used to improve cache behavior.
- The innermost loop (loop index changes fastest) should (only) index the right-most array index expression in case of row-major storage like in C.
- Can improve execution time by 1 or 2 orders of magnitude.

Loop Interchange (cont’d)

- Loop interchange can also expose parallelism.
- If an inner-loop does not carry a dependence (entry in direction vector equals ‘=’), this loop can be executed in parallel.
- Moving this loop outwards increases the granularity of the parallel loop iterations.
Loop Interchange:
Example of locality improvement

for (i=0; i<N; i++)
for (j=0; j<M; j++)
B[i][j] = f(A[j],B[i][j-1]);

for (j=0; j<M; j++)
for (i=0; i<N; i++)
B[i][j] = f(A[j],B[i][j-1]);

- Productions and consumptions as local and regular as possible
- Exploration required

Loop Interchange:
Satisfy dependencies

- No loop-carried dependencies allowed unless \( \forall I_2: \prod(I_2) < \text{cons}(I_2) \)

for (i=0; i<N; i++)
for (j=0; j<M; j++)
A[i][j] = f(A[i-1][j+1]);

for (j=0; j<M; j++)
for (i=0; i<N; i++)
A[i][j] = f(A[i-1][j]);

- Enablers:
  - data-flow transformations
  - Loop Bump
Loop Skew: Basic transformation

for $l_1 = \exp_1$ to $\exp_2$
for $l_2 = \exp_3$ to $\exp_4$
$A(l_1, l_2)$

for $l_1 = \exp_1$ to $\exp_2$
for $l_2 = \exp_3 + \alpha.l_1 + \beta$ to $\exp_4 + \alpha.l_1 + \beta$
$A(l_1, l_2 - \alpha.l_1 - \beta )$

Loop Skew

for $(j=0; j<H; j++)$
for $(i=0; i<W; i++)$
$A[i][j] = ...$

for $(j=0; j<H; j++)$
for $(i=0+j; i<W+j; i++)$
$A[I-j][j] = ...$
Loop Skew: Example as enabler of regularity improvement

\[
\begin{align*}
\text{for } (i=0; i<N; i++) \\
\text{for } (j=0; j<M; j++) \\
... &= f(A[i+j]);
\end{align*}
\]

⇒

\[
\begin{align*}
\text{for } (i=0; i<N; i++) \\
\text{for } (j=i; j<i+M; j++) \\
... &= f(A[j]);
\end{align*}
\]

Loop Extend & Interchange

⇒

\[
\begin{align*}
\text{for } (j=0; j<N+M; j++) \\
\text{for } (i=0; i<N; i++) \\
\text{if } (j>=i \&\& j<i+M) \\
... &= f(A[j]);
\end{align*}
\]

Loop Tiling

\[
\begin{align*}
\text{for } I = 0 \text{ to } \exp_1 \cdot \exp_2 \\
A(I)
\end{align*}
\]

\[
\begin{align*}
\text{Tile size } \exp_1 \\
\text{Tile factor } \exp_2
\end{align*}
\]

⇒

\[
\begin{align*}
\text{for } I_1 = 0 \text{ to } \exp_2 \\
\text{for } I_2 = \exp_1 \cdot I_1 \text{ to } \exp_1 \cdot (I_1 + 1) \\
A(I_2)
\end{align*}
\]
Loop Tiling

- Improve cache reuse by dividing the iteration space into tiles and iterating over these tiles.
- Only useful when working set does not fit into cache or when there exists much interference.
- Two adjacent loops can legally be tiled if they can legally be interchanged.
2-D Tiling Example

```c
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        A[i][j] = B[j][i];
```

```c
for(TI=0; TI<N; TI+=16)
    for(TJ=0; TJ<N; TJ+=16)
        for(i=TI; i<min(TI+16,N); i++)
            for(j=TJ; j<min(TJ+16,N); j++)
                A[i][j] = B[j][i];
```

## Selecting a Tile Size

- Current tile size selection algorithms use a cache model:
  - Generate collection of tile sizes;
  - Estimate resulting cache miss rate;
  - Select best one.
- Only take into account L1 cache.
- Mostly do not take into account n-way associativity.
Loop Index Split

\[ \text{for } l_a = \exp_1 \text{ to } \exp_2 \]
\[ A(l_a) \]

\[ \Rightarrow \]

\[ \text{for } l_a = \exp_1 \text{ to } p \]
\[ A(l_a) \]

\[ \text{for } l_b = p+1 \text{ to } \exp_2 \]
\[ A(l_b) \]

Loop Unrolling

- Duplicate loop body and adjust loop header.
- Increases available ILP, reduces loop overhead, and increases possibilities for common subexpression elimination.
- Always valid.
Loop Unrolling: Downside

- If unroll factor is not divisor of trip count, then need to add remainder loop.
- If trip count not known at compile time, need to check at runtime.
- Code size increases which may result in higher I-cache miss rate.
- Global determination of optimal unroll factors is difficult.

(Partial) Loop Unrolling

```
for l = \exp_1 to \exp_2 \\
A(l)
```

\[\Rightarrow\]

```
A(\exp_1) \\
A(\exp_1 + 1) \\
A(\exp_1 + 2) \\
... \\
A(\exp_2)
```

```
for l = \exp_1/2 to \exp_2/2 \\
A(2l) \\
A(2l+1)
```
Loop Unroll: Example of removal of non-affine iterator

for (L=1; L < 4; L++)
  for (i=0; i < (1<<L); i++)
    A(L,i);

for (i=0; i < 2; i++)
  A(1,i);
for (i=0; i < 4; i++)
  A(2,i);
for (i=0; i < 8; i++)
  A(3,i);

Unroll-and-Jam

- Unroll outerloop and fuse new copies of the innerloop.
- Increases size of loop body and hence available ILP.
- Can also improve locality.
Unroll-and-Jam Example

for (i=0;i<N;i++)
for (j=0;j<N;j++)
A[i][j] = B[j][i];

for (i=0; i<N; i+=2)
for (j=0; j<N; j++) {
  A[i][j] = B[i][i];
  A[i+1][j] = B[j][i+1];
}

- More ILP exposed
- Spatial locality of $B$ enhanced

Simplified loop transformation script

- Give all loops same nesting depth
  - Use one-iteration dummy loops if necessary
- Improve regularity
  - Usually applying loop interchange or reverse
- Improve locality
  - Usually applying loop merge
  - Break data dependencies with loop bump/skew
  - Sometimes loop index split or unrolling is easier
Loop transformation theory

- Iteration spaces
- Polyhedral model
- Dependency analysis

Technical Preliminaries (1)

\[
\text{do } i = 2, N \\
\hspace{1em} \text{do } j = i, N \\
\hspace{2em} x[i,j] = 0.5 \times (x[i-1,j] + x[i,j-1]) \\
\text{enddo}
\]

- Address expr \((1)\) \[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
i \\
j
\end{bmatrix}
+ \\
\begin{bmatrix}
-1 \\
0
\end{bmatrix}
\]

- perfect loop nest
- iteration space
- dependence vector
Technical Preliminaries (2)

Switch loop indexes:
\[
do \ m = 2, N \\
\quad do \ n = 2, m \\
\quad \quad x[n,m] = 0.5 \times (x[n-1,m] + x[n,m-1])
\]

\[ (2) \]

\[
\begin{bmatrix} m \\ n \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

\[
\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}
\]

affine transformation

Polyhedral Model

- Polyhedron is set \( \{ x : Ax \leq c \} \) for some matrix \( A \) and bounds vector \( c \)
- Polyhedra (or Polytopes) are objects in a many-dimensional space without holes
- Iteration spaces of loops (with unit stride) can be represented as polyhedra
- Array accesses and loop transformations can be represented as matrices
Iteration Space

- A loop nest is represented as $Bl \leq b$ for iteration vector $l$
- Example:

```
for(i=0; i<10; i++)
  for(j=i; j<10; j++)
    A[j][i]
```

- $A \leftrightarrow \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 0 \\ 0 & 1 & 9 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} b \end{bmatrix}$

Array Accesses

- Any array access $A[e1][e2]$ for linear index expressions $e1$ and $e2$ can be represented as an access matrix and offset vector.
  
  $A + a$

- This can be considered as a mapping from the iteration space into the storage space of the array (which is a trivial polyhedron)
Unimodular Matrices

- A unimodular matrix $T$ is a matrix with integer entries and determinant $\pm 1$.
- This means that such a matrix maps an object onto another object with exactly the same number of integer points in it.
- Its inverse $T^{-1}$ always exist and is unimodular as well.

Types of Unimodular Transformations

- Loop interchange
- Loop reversal
- Loop skewing for arbitrary skew factor
- Since unimodular transformations are closed under multiplication, any combination is a unimodular transformation again.
Application

- Transformed loop nest is given by $AT^1 I' \leq a$
- Any array access matrix is transformed into $AT^1$.
- Transformed loop nest needs to be normalized by means of Fourier-Motzkin elimination to ensure that loop bounds are affine expressions in more outer loop indices.

Dependence Analysis

Consider following statements:

$S1: a = b + c;$
$S2: d = a + f;$
$S3: a = g + h;$

- $S1 \rightarrow S2$: true or flow dependence = RaW
- $S2 \rightarrow S3$: anti-dependence = WaR
- $S1 \rightarrow S3$: output dependence = WaW
Dependences in Loops

- Consider the following loop
  
  ```
  for(i=0; i<N; i++){
    S1: a[i] = ...;
    S2: b[i] = a[i-1];
  }
  ```

- Loop carried dependence S1 → S2.
- Need to detect if there exists i and i’ such that i = i’-1 in loop space.

Definition of Dependence

- There exists a dependence if there two statement instances that refer to the same memory location and (at least) one of them is a write.
- There should not be a write between these two statement instances.
- In general, it is undecidable whether there exist a dependence.
Direction of Dependence

- If there is a flow dependence between two statements $S_1$ and $S_2$ in a loop, then $S_1$ writes to a variable in an earlier iteration than $S_2$ reads that variable.
- The iteration vector of the write is lexicographically less than the iteration vector of the read.
- $I \preceq I'$ iff $i_1 = i'_1 \land \ldots \land i_{(k-1)} = i'_{(k-1)} \land ik < i'k$ for some $k$.

Direction Vectors

- A direction vector is a vector
  
  \[ (=,=,\ldots,=,<,*,*\ldots,*\) \]
  
  - where * can denote $=$ or $<$ or $>$.
- Such a vector encodes a (collection of) dependence.
- A loop transformation should result in a new direction vector for the dependence that is also lexicographically positive.
Loop Interchange

- Interchanging two loops also interchanges the corresponding entries in a direction vector.
- Example: if direction vector of a dependence is \(<,>\) then we may not interchange the loops because the resulting direction would be \(>,<\) which is lexicographically negative.

Affine Bounds and Indices

- We assume loop bounds and array index expressions are affine expressions:
  \[ a_0 + a_1 \times i_1 + \ldots + a_k \times i_k \]
- Each loop bound for loop index \(i_k\) is an affine expressions over the previous loop indices \(i_1\) to \(i(k-1)\).
- Each loop index expression is a affine expression over all loop indices.
Non-Affine Expressions

- Index expressions like \( i \times j \) cannot be handled by dependence tests. We must assume that there exists a dependence.
- An important class of index expressions are indirections \( A[B[i]] \). These occur frequently in scientific applications (sparse matrix computations).
- In embedded applications???

Linear Diophantine Equations

- A linear diophantine equations is of the form
  \[
  \sum a_j \times x_j = c
  \]
- Equation has a solution iff \( \gcd(a_1,\ldots,a_n) \) is divisor of \( c \)
GCD Test for Dependence

- Assume single loop and two references A[a+bi] and A[c+di].
- If there exist a dependence, then \( \gcd(b,d) \) divides \( (c-a) \).
- Note the direction of the implication!
- If \( \gcd(b,d) \) does not divide \( (c-a) \) then there exists no dependence.

GCD Test (cont’d)

- However, if \( \gcd(b,d) \) does divide \( (c-a) \) then there might exist a dependence.
- Test is not exact since it does not take into account loop bounds.
- For example:
  - for(i=0; i<10; i++)
  - \( A[i] = A[i+10] + 1; \)
GCD Test (cont’ d)

- Using the Theorem on linear diophantine equations, we can test in arbitrary loop nests.
- We need one test for each direction vector.
- Vector (\(=,=,\ldots,=,<,\ldots\)) implies that first k indices are the same.
- See book by Zima for details.

Other Dependence Tests

- There exist many dependence test
  - Separability test
  - GCD test
  - Banerjee test
  - Range test
  - Fourier-Motzkin test
  - Omega test
- Exactness increases, but so does the cost.
Fourier-Motzkin Elimination

- Consider a collection of linear inequalities over the variables $i_1, \ldots, i_n$
  - $e_1(i_1, \ldots, i_n) \leq e_1'(i_1, \ldots, i_n)$
  - $\ldots$
  - $e_m(i_1, \ldots, i_n) \leq e_m'(i_1, \ldots, i_n)$

- Is this system consistent, or does there exist a solution?
- FM-elimination can determine this.

FM-Elimination (cont’d)

- First, create all pairs $L(i_1, \ldots, i(n-1)) \leq i_n \leq U(i_1, \ldots, i(n-1))$. This is solution for $i_n$.
- Then create new system
  - $L(i_1, \ldots, i(n-1)) \leq U(i_1, \ldots, i(n-1))$
  - together with all original inequalities not involving $i_n$.
- This new system has one variable less and we continue this way.
FM-Elimination (cont’ d)

- After eliminating i1, we end up with collection of inequalities between constants \( c_1 \leq c_1' \).
- The original system is consistent iff every such inequality can be satisfied.
- There does not exist an inequality like
  \[
  10 \leq 3.
  \]
- There may be exponentially many new inequalities generated!

Fourier-Motzkin Test

- Loop bounds plus array index expressions generate sets of inequalities, using new loop indices \( i' \) for the sink of dependence.
- Each direction vector generates inequalities
  \[
  i_1 = i_1' \ldots i(k-1) = i(k-1)' \quad ik < ik'
  \]
- If all these systems are inconsistent, then there exists no dependence.
- This test is not exact (real solutions but no integer ones) but almost.
N-Dimensional Arrays

- Test in each dimension separately.
- This can introduce another level of inaccuracy.
- Some tests (FM and Omega test) can test in many dimensions at the same time.
- Otherwise, you can linearize an array: Transform a logically N-dimensional array to its one-dimensional storage format.

Hierarchy of Tests

- Try cheap test, then more expensive ones:
- if (cheap test1 = NO)
  - then print ‘NO’
- else if (test2 = NO)
  - then print ‘NO’
- else if (test3 = NO)
  - then print ‘NO’
- else …
Practical Dependence Testing

- Cheap tests, like GCD and Banerjee tests, can disprove many dependences.
- Adding expensive tests only disproves a few more possible dependences.
- Compiler writer needs to trade-off compilation time and accuracy of dependence testing.
- For time critical applications, expensive tests like Omega test (exact!) can be used.