Optimizing Loop Transformations

and data dependence analysis for loops

Why Loop Optimizations?

Loops are a promising object for compiler optimizations:

- High execution frequency
- Most computation done in (inner) loops
- Even small optimizations can have large impact
- Regular, repetitive behavior
- Compact description
- Relatively simple to analyze statically
- Well researched

Loop Optimizations – General Issues

- Move loop invariant computations out of loops
- Modify the order of iterations or parts thereof

Goals:

- Improve data access locality
- Faster execution
- Reduce loop control overhead
- Enhance possibilities for loop parallelization or vectorization

Only transformations that preserve the program semantics (its input/output behavior) are admissible

- Conservative (static) criterion: preserve data dependences
- Need data dependence analysis for loops

Loop Invariant Code Hoisting

- Move loop invariant code out of the loop:

Example:

```c
for (i=0; i<10; i++)
    a[i] = b[i] + c / d;
```

Move loop invariant code out of the loop:

```c
tmp = c / d;
for (i=0; i<10; i++)
    a[i] = b[i] + tmp;
```

Loop Unrolling

- Loop unrolling

Example:

```c
for (i=0; i<50; i++)
    a[i] = b[i];
```

Unroll loop:

```c
for (i =0; i<50; i+=2)
    a[i] = b[i];
a[i+1] = b[i+1];
```

- Reduces loop overhead (total # comparisons, branches, increments)
- May enable further local optimizations
- Drawback: longer code

Exercise: Formulate the unrolling rule for statically unknown upper loop limit

Loop Interchange (1)

- For properly nested loops (statements in innermost loop body only)

Example 1:

```c
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        a[i][j] = 0.0;
```

Interchange inner loops:

```c
for (j=0; j<M; j+)
    for (i=0; i<N; i++)
        a[i][j] = 0.0;
```

- Can improve data access locality (fewer cache misses / page faults)
Foundations: Loop-Carried Data Dependences

- Recall:
  - Data dependence \( S \to T \), if operation \( S \) may execute (dynamically) before operation \( T \) and both may access the same memory location, and at least one of these accesses is a write.
  - In general, only a conservative over-estimation can be determined statically.
  - Data dependence \( S \to T \) is called loop carried by a loop \( L \) if the data dependence \( S \to T \) may exist for instances of \( S \) and \( T \) in different iterations of \( L \).

Example:

\[
\begin{align*}
L: \quad & \text{for } (i=1; i<N; i++) \\
\quad & \{ \\
\quad & \quad \text{if } i \equiv 0 \mod 4 \{ \\
\quad & \quad \quad S_1: \quad \ldots = x[i-1]; \\
\quad & \quad \quad S_2: \quad \ldots = x[i]; \\
\quad & \quad \} \\
\quad & \quad \text{endif} \\
\quad & \} \\
\end{align*}
\]

\( \to \) partial order between the operation instances resp. iterations.

Loop Interchange (2)

- Be careful with loop carried data dependences!
- Example 2:

\[
\begin{align*}
f(i=1; j=M; j++) & \quad \text{for } (i=1; i<N; i++) \\
& \quad \{ \\
& \quad \quad S_1: \quad \ldots = a[i][j]; \\
& \quad \quad S_2: \quad \ldots = a[i][j+1]; \\
& \quad \} \\
\end{align*}
\]

Iteration space:

\[
\begin{align*}
& \text{old iteration order} \\
& \text{new iteration order}
\end{align*}
\]

- Interchanging the loop headers would violate the partial iteration order given by the data dependences.

Loop Fusion

- Merge subsequent loops with same header
  - Example:

\[
\begin{align*}
& \text{for } (i=1; j=M; j++) \\
& \quad \{ \\
& \quad \quad S_1: \quad \ldots = a[i][j]; \\
& \quad \quad S_2: \quad \ldots = a[i][j+1]; \\
& \quad \} \\
\end{align*}
\]

Iteration space:

\[
\begin{align*}
& \text{old iteration order} \\
& \text{new iteration order}
\end{align*}
\]

- Generally: Interchanging loop headers is only admissible if loop-carried dependences have the same direction for all loops in the loop nest (all directed along or all against the iteration order).

Data Dependence Analysis — Overview

- Important for loop optimizations, vectorization and parallelization, instruction scheduling, data cache optimizations
  - Conservative approximations to disjointness of pairs of memory accesses
    - weaker than data-flow analysis
    - but generalizes nicely to the level of individual array element
- Loops, loop nests
  - Iteration space
  - Array subscripts in loops
  - Index space
- Dependence testing methods
- Data dependence graph
- Data + control dependence graph
- Program dependence graph
Precedence relation between statements

- $S_1$ statically (textually) precedes $S_2$ : $S_1 \text{ pred } S_2$
- $S_1$ dynamically precedes $S_2$ : $S_1 \preceq S_2$

```plaintext
S1: s ← 0
   for / from 1 to n do
     S2: x ← s + a[i]
     S3: a[i] ← x
   od
```

Control and Data Dependence; Dependence Graph

Dependence = constraint on execution order

- control dependence by control flow: $S_i \rightarrow S_j$
- data dependence:
  - flow / true dependence: $S_i \rightarrow S_j$
  - $S_i < S_j$ and $\exists x : S_i$ writes $x$, $S_j$ reads $x$
  - anti-dependence: $S_i \rightarrow S_j$
  - $S_i < S_j$ and $\exists x : S_i$ reads $x$, $S_j$ writes $x$
- output dependence: $S_i \rightarrow S_j$
  - $S_i < S_j$ and $\exists x : S_i$ writes $x$, $S_j$ writes $x$

Data Dependence Graph

- Data dependence graph: directed graph
- States, precedence constraints due to data dependences
- For basic blocks:
  - Acyclic dependence graph (DAG)
- Within loops, loop nests: $\text{pred } \neq \preceq$
  - Edge if dependence may exist for some pair of iterations
  - Cycles possible
  - Loop-independent versus loop-carried dependences

Loop Normalization

Given a loop of the form
```
for / from 1 to n step S do
  ...f...
od
```
- Normalize the loop:
  - Lower bound 0 (C) resp. 1 (Fortran)
  - Step size +1
- Update all occurrences of the loop counter $l$ by $l = l + S + L$

```
S1: s ← 0
   for / from 1 to n do
     S2: x ← s + a[i]
     S3: a[i] ← x
   od
```

Remark: Target-Level Dependence Graphs

- For VLIR / target code, the dependence edges may be labeled with latency or delay of the operation
- See lecture on instruction scheduling for details

Loop Iteration Space

Beyond basic blocks: $\text{pred } \neq \preceq$

- Canonical loop nest: (HIR code)
  - for $i_1$ from 1 to $n_1$ do
    - for $i_2$ from 1 to $n_2$ do
      - $\ldots$
      - $\ldots$

- Iteration space: $\text{BS} = [1..n] \times [1..n] \times \ldots \times [1..n]$
  - (simplest case: rectangular, static loop bounds)
- Iteration vector $\bar{i} = (i_1, i_2, \ldots, i_n) \in \text{BS}
Dependence Distance and Direction

Lexicographic order on iteration vectors → dynamic execution order:

\[ S_i((i_1, \ldots, i_n)) < S_j((j_1, \ldots, j_n)) \iff \]

- either \( S_i \) pred. \( S_j \) and \( (i_1, \ldots, i_n) \) \( \leq \) \( (j_1, \ldots, j_n) \)
- or \( S_j = S_i \) and \( (j_1, \ldots, j_n) <_{\text{lex}} (i_1, \ldots, i_n) \)

**distance vector** \( d = j - i = (j_1 - i_1, \ldots, j_n - i_n) \)

**direction vector** \( \text{dirv} = \text{sgn}(d) = (\text{sgn}(d_1), \ldots, \text{sgn}(d_n)) \)

in terms of symbols \( = < \leq \) ≥

Example: \( S_i((i_1, i_2, i_3)) \Rightarrow S_j((j_1, j_2, j_3)) \)

- distance vector \( d = (0, 0, 1) \), direction vector \( \text{dirv} = (0, 0, 1) \)
- loop-independent dependence

Example: \( S_i((i_1, i_2, i_3)) \Rightarrow S_j((j_1, j_2, j_3)) \)

- distance vector \( d = (0, 0, 0) \), direction vector \( \text{dirv} = (0, 0, 0) \)
- loop-carried dependence (carried by \( i \)-loop at level 3)

Dependence Equation System

One-dimensional array \( i \) accessed in \( i \) nested loops:

\[ S_i: \quad \ldots \text{A}(\text{f}(i)) \ldots \]
\[ S_j: \quad \ldots \text{A}(\text{g}(j)) \ldots \]

Is there a dependence between \( S_i(i) \) and \( S_j(j) \) for some \( i, j \in b \)?

Typically, \( f, g \) linear:

\[ f(i) = a_0 + \sum a_i i \]
\[ g(j) = b_0 + \sum b_i j \]

Exist \( i, j \in b \) such that \( f(i) = g(j) \), i.e.,

\[ a_0 + \sum a_i i = b_0 + \sum b_i j \]

**der. equator** subject to \( i, j \in b \), i.e.,

\[ 1 \leq i \leq n_1, \quad 1 \leq j \leq n_2 \]

\[ 1 \leq i \leq n_1, \quad 1 \leq j \leq n_2 \]

\[ \Rightarrow \text{constrained linear Diophantine equation system} \quad \rightarrow \text{ILP (NP-complete)} \]

Linear Diophantine Equations

\[ \sum_{j=1}^{k} a_j x_j = c \]

where \( a \geq 1, \quad c, a_j \in \mathbb{Z}, \quad \exists j: a_j \neq 0, \quad x_i \in \mathbb{Z} \)

Example 1: \( x + 4y = 1 \)

has infinitely many solutions, e.g., \( x = 5 \) and \( y = -1 \).

Example 2: \( 5x - 10y = 2 \)

has no solution in \( \mathbb{Z} \); absolute term must be multiple of

**Theorem:**

\[ \sum_{j=1}^{k} a_j x_j = c \quad \text{has a solution if and only if} \quad \gcd(a_1, \ldots, a_k) \mid c. \]

**Proof:** see e.g. [Zima/Chapman p. 143]

Dependence Testing, 1: GCD-Test

Often, a simple test is sufficient to prove independence: e.g.,

**gcd-test** [Banerjee'/76], [Towlie'/76]:

independence if

\[ \gcd \left( \bigcup_{i=1}^{k} (a_i, b_i) \right) \mid \sum_{i=0}^{k} (a_i - b_i) \]

**constraints on \( i \)'s not considered**

Example:

for \( i \) from 1 to 4 do

\[ S_1: \quad b[i] \leftarrow a[3 \ast i - 5] + 2 \]
\[ S_2: \quad a[2 \ast i + 1] \leftarrow 1.0 \ast i \]

solution to \( 2i + 1 = 3j - 5 \) exists in \( \mathbb{Z} \) as \( \gcd(3, 2) \mid (-5 - 1 + 3 - 2) \)

not checked whether such \( i, j \) exist in \( \{1, \ldots, 4\} \)

For multidimensional arrays?

**subscript-wise test vs. linearized indexing**

for \( i \) ...

\[ S_1: \quad a[i, j, k] \leftarrow \ldots \]
\[ S_2: \quad a[i, j, k] \leftarrow \ldots \]

Moreover:

Hierarchical structuring of dependence tests [Burke/Cytron'/66]
Survey of Dependence Tests

gcd test
separability test (gcd test for special case, exact)
Banerjee-Wolfe test [Banerjee’89] rational solution in $I/S$
Delta-test [Goff/Kennedy/Tseng’91]
Power test [Wolfe/Tseng’91]
Simple Loop Residue test [Maydan/Hennessy/Lam’91]
Fourier-Motzkin Elimination [Maydan/Hennessy/Lam’91]
Omega test [Pugh/Wonnacott’92]

Loop Parallelization

A transformation that reorders the iterations of a level-i-loop, without making any other changes, is valid if the loop carries no dependence.

for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        for (k=1; k<r; k++)
            $s_i: a[i][j][k] = \ldots a[i][j-1][k] \ldots$ $(+,<,=)$

It is valid to convert a sequential loop to a parallel loop if it does not carry a dependence.

for (i=1; i<n; i++) $\rightarrow$ forall (i, l, n, p)
    $s_i: b[i] = 2 * c[i];$

Loop Transformations

Goal

- modify execution order of loop iterations
- preserve data dependence constraints

Motivation

- data locality
  - increase reuse of registers, cache
- parallelism
  - eliminate loop-carried dependences, increase granularity
- reduced overhead

Loop Distribution (a.k.a. Loop Fission)

for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        for (k=1; k<r; k+)
            $s_i: a[i][j][k] = a[i][j][k] * k;$

Safe if all statements forming a SCC in the dependence graph end up in the same loop.
Forward (loop-carried) dependences are okay, but keep topological order.
- often enables vectorization; better cache utilization of each loop.

Loop Fusion

for $i$ from 1 to N do
    $a[i] \leftarrow a[i] + H[i];$
for $j$ from 1 to N do
    $a[j] \leftarrow a[j] * c[j];$

For arrays large enough, $a[j]$ will no longer be cached.

Safe if neither loop carries a (backward) dependence.
- locality can be converted inter-loop reuse to intra-loop reuse
- larger basic blocks
- reduce loop overhead

Strip Mining / Loop Blocking / -Tiling

for (i=0; i<n; i++)
    $a[i] = b[i] + c[i];$

$\uparrow$ loop blocking with block size $k$
for (i=0; i<n; i++) // loop over blocks
    for (i=0; i<2^h; i++) // loop within blocks
        $a[i][i] = b[i][i] + c[i][i];$

Tiling = blocking in multiple dimensions + loop interchange

Goal: increase locality; support vectorization (vector registers)
Reverse transformation: Loop linearization
Tiled Matrix-Matrix Multiplication (1)

Matrix-Matrix multiplication \( C = A \times B \) here for square \((n \times n)\) matrices \(C, A, B\), with \(n\) large \((-10^3)\):

- \(C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}\) for all \(i, j = 1...n\)
- Standard algorithm for Matrix-Matrix multiplication (here without the initialization of \(C\)-entries to 0):
  
  ```
  for (i=0; i<n; i++)
  for (j=0;  j<n;  j++)
  for (k=0;  k<n;  k++)
      C[i][j] += A[i][k] * B[k][j];
  ```

Good spatial locality on \(A, C\)
Bad spatial locality on \(B\) (many capacity misses)

Tiled Matrix-Matrix Multiplication (2)

- Block each loop by block size \(S\) (choose \(S\) so that a block of \(A, B, C\) fit in cache together), then interchange loops

Code after tiling:

```
for (ii=0; ii<n; ii+=S)
for (jj=0;  jj<n;  jj+=S)
for (kk=0;  kk<n;  kk+=S)
    for (i=ii;  i < ii+S;  i++)
    for (j=jj;  j < jj+S;  j++)
    for (k=kk;  k < kk+S;  k++)
        C[i][j] += A[i][k] * B[k][j];
```

Good spatial locality for \(A, B\) and \(C\)

Remark on Locality Transformations

- An alternative can be to change the data layout rather than the control structure of the program

  **Example:** Store matrix \(B\) in transposed form, or, if necessary, consider transposing it, which may pay off over several subsequent computations
  - Finding the best layout for all multidimensional arrays is a NP-complete optimization problem [Mace, 1988]

  **Example:** Recursive array layouts that preserve locality
  - Morton-order layout
  - Hierarchically tiled arrays

Loop Nest Flattening / Linearization

Flattens a multidimensional iteration space to a linear space:

```
for \(i\) from 0 to \(m-1\) do
    \(i \leftarrow k \mod m\)
    \(j \leftarrow k \div m\)
    iteration(i,j)
```

+ larger iteration space, better for scheduling / load balancing
  - overhead to reconstruct original iteration variables may be reduced by using induction variables \(i, j\) that are updated by accumulating additions instead of div and mod

Loop Unrolling

```for \(i\) from 1 to 100 do
    \(a[i] \leftarrow a[i] + b[i]\)
    \(a[i+1] \leftarrow a[i+1] + b[i+1]\)
    \(a[i+2] \leftarrow a[i+2] + b[i+2]\)
    \(a[i+3] \leftarrow a[i+3] + b[i+3]\)
```  

Unroll by 4:

```for \(i\) from 1 to 100 step 4 do
    \(a[i] \leftarrow a[i] + b[i]\)
    \(a[i+1] \leftarrow a[i+1] + b[i+1]\)
    \(a[i+2] \leftarrow a[i+2] + b[i+2]\)
    \(a[i+3] \leftarrow a[i+3] + b[i+3]\)
```

+ less overhead per useful operation
+ longer basic blocks for local optimizations
  - (local CSE, local reg.-allocation, local scheduling, SW pipelining)
+ longer code

Loop Unrolling with Unknown Upper Bound

```for \(i\) from 1 to \(N\) do
    \(a[i] \leftarrow a[i] + b[i]\)
```

Unroll by 4:

```for \(i\) from 1 to \(N\) do
    \(a[i] \leftarrow a[i] + b[i]\)
    \(a[i+1] \leftarrow a[i+1] + b[i+1]\)
    \(a[i+2] \leftarrow a[i+2] + b[i+2]\)
    \(a[i+3] \leftarrow a[i+3] + b[i+3]\)
```

Used e.g. in BLAS
**Loop Unroll-And-Jam**

unroll the outer loop and fuse the resulting inner loops:

```plaintext
for i from 1 to N do
  a[i] ← a[i] + b[i]
od

for j from 1 to N do
  a[i] ← a[i] + b[j]
  a[i] ← a[i] + b[j]
  od

unroll & jam:
for i from 1 to N step 2 do
  a[i] ← a[i] + b[i]
  a[i+1] ← a[i+1] + b[i]
od
```

The same conditions as for loop interchange (for the two innermost loops after the unrolling step) must hold.

(See: [Allen/Kennedy 02, Ch. 8.4.1]).

- increases reuse in inner loop
- less overhead

---

**Loop Peeling**

remove the first (or last) iteration of the loop
and clone the loop body for that iteration:

```plaintext
if N ≥ 1 then
  a[i] ← (x+y) * b[i]
  od

for i from 1 to N do
  a[i] ← (x+y) * b[i]
  od
```

(Test on trip count can be removed if N ≥ 1 is statically known.)

+ can enable loop fusion
+ may extract conditionals handling boundary cases from the loop
- longer code

---

**Index Set Splitting**

Divide the iteration space into two portions.

```plaintext
for i from 1 to 100 do
  a[i] ← b[i] + c[i]
  od

if i > 10 then
  a[i] ← a[i] + a[i-10]
  split after 10:
  for i from 11 to 100 do
    a[i] ← b[i] + c[i]
    od

  a[i] ← a[i] + a[i-10]
  od
```

+ removes condition evaluation in every iteration
+ factors out the parallelizable set of iterations
- longer code

---

**Loop Unswitching**

if expression then
  for i from 1 to 100 do
    a[i] ← a[i] + b[i]
    od
  unswitch:
  if else
    for i from 1 to 100 do
      a[i] ← a[i] + b[i]
      od
  fi

+ hoist loop-invariant control flow out of loop nest
+ no tests, no branches in loop body
- larger basic blocks (see above), simpler software pipelining
- longer code

---

**Scalar Replacement**

For (inner) loops accumulating a value in an array element
use a temporary scalar for the accumulator variable:

```plaintext
for i from 1 to N do
  t ← a[i]
  od

for j from 1 to N do
  a[j] ← a[j] + b[i]
  scalar repl.:
  od
```

+ keep i in a register all the time
+ saves many costly memory accesses to a[i]

---

**Scalar Expansion / Array Privatization**

promote a scalar temporary to an array to break a dependence cycle:

```plaintext
allocate r[1..N]
for i from 1 to N do
  r[i] ← a[i] + b[i]
  c[i] ← t + 1
  expand scalar:
  r[i] ← r[i] + 1
  od
```

- removes the loop-carried antidependence due to r
- can now parallelize the loop!
- needs more array space

Loop must be countable, scalar must not have upward exposed uses.

May also be done conceptually only, to enable parallelization:

Just create one private copy of i for every processor + array privatization

```plaintext
allocate i[1..N]
for i from 1 to N do
  t ← r[i]
  od
```
Outlook: Runtime Parallelization

Sometimes parallelizability cannot be decided statically.

```c
if is_parallelizable(...) 
  forall i in [0..n-1] do   // parallel version of the loop
    iteration(i);
  od
else
  for i from 0 to n-1 do   // sequential version of the loop
    iteration(i);
  od
fi
```

The runtime dependence test `is_parallelizable(...)` itself may partially run in parallel.

Outlook: Parallelization by Pattern Matching

Traditional loop parallelization fails for loop-carried dep. with distance 1:

```c
S0: s = 0;
    for (i=1; i<n; i++)
S1:    s = s + a[i];
S2: a[0] = c[0];
    for (i=1; i<n; i++)
S3:    a[i] = a[i-1] * b[i] + c[i];
```

- `# idiom recognition (pattern matching)`

```
S1': s = VSUM(a[1:n-1], 0);
S3': a[0:n-1] = FOLM(b[1:n-1], c[0:n-1], mul, add);
```

- `Algorithm replacement`

```
S1'': s = par_sum(a, 0, n, 0);
```