A Crash Course in Compilers for Parallel Computing

Mary Hall

Fall, 2008
Overview of “Crash Course”

- L1: Data Dependence Analysis and Parallelization (Oct. 30)
- L2 & L3: Loop Reordering Transformations, Reuse Analysis and Locality Optimization (Nov. 6)
- L4: Autotuning Compiler Technology (Nov. 13)
Outline of Lecture

I. Summary of Last Week
II. Reuse Analysis
III. Two Loop Reordering Transformations
   - Permutation
   - Tiling (aka blocking)
IV. Locality Optimization
I. Summary: Data Dependence

True (flow) dependence
\[ a = a \]

Anti-dependence
\[ a = a \]

Output dependence
\[ a = a \]

Input dependence (for locality)
\[ a = a \]

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 \( i' \)
Restrict to an Affine Domain

for (i=1; i<N; i++)
    for (j=1; j<N j++) {
        A[i+2*j+3, 4*i+2*j, 3*i] = ...;
        ... = A[1, 2*i+1, j];
    }

• Only use loop bounds and array indices which are integer linear functions of loop variables.

• Non-affine example:
  for (i=1; i<N; i++)
      for (j=1; j<N j++) {
          A[i*j] = A[i*(j-1)];
      }
Distance Vectors

N = 6;
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        A[i+1,j+1] = A[i,j] * 2.0;

• Distance vector = \([1,1]\)

• A loop has a distance vector \(D\) if there exists data dependence from iteration vector \(I\) to a later vector \(I'\), and \(D = I' - I\).

• Since \(I' > I\), \(D \geq 0\).
  (\(D\) is lexicographically greater than or equal to 0).
Equivalence to Integer Programming

- Need to determine if $F(i) = G(i')$, where $i$ and $i'$ are iteration vectors, with constraints $i, i' \geq L, U \geq i, i'$

- Example:

```plaintext
for (i=2; i<=100; i++)
    A[i] = A[i-1];
```

- Inequalities:

$$
\begin{bmatrix}
-1 & 0 \\
1 & 0 \\
-1 & 1 \\
1 & -1 \\
0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
i_1 \\
i_2 \\
\end{bmatrix}
\leq
\begin{bmatrix}
0 \\
100 \\
-1 \\
1 \\
100 \\
\end{bmatrix}
$$

Solution exist?
Yes $\Rightarrow$ dependence
Fundamental Theorem of Dependence

• **Theorem 2.2:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
II. Introduction to Locality Optimization and Reuse Analysis

- Large memories are slow, fast memories are small
- Hierarchy allows fast and large memory on average
- Managing *locality* crucial for achieving high performance
Cache basics: a quiz

- **Cache hit:**
  - in-cache memory access—cheap

- **Cache miss:**
  - non-cached memory access—expensive
  - need to access next, slower level of hierarchy

- **Cache line size:**
  - # of bytes loaded together in one entry
  - typically a few machine words per entry

- **Capacity:**
  - amount of data that can be simultaneously in cache

- **Associativity**
  - direct-mapped: only 1 address (line) in a given range in cache
  - *n*-way: \( n \geq 2 \) lines w/ different addresses can be stored
How do we get locality (in caches)?

• Data locality:
  - data is reused and is present in cache
  - same data or same cache line

• Data reuse:
  - data used multiple times
  - intrinsic in computation

• If a computation has reuse, what can we do to get locality?
  - code reordering transformations (today)
  - data layout
Temporal Reuse

• Same data used in distinct iterations I and I’

```c
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
```

• \(A[j]\) has self-temporal reuse in loop i
Spatial Reuse

- Same cache line used in distinct iterations \( I \) and \( I' \)

```c
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
```

- \( A[j] \) has self-spatial reuse in loop \( j \)

- **Multi-dimensional array note:** C arrays are stored in row-major order, while FORTRAN arrays are stored in column-major order)
Group Reuse

• Same data used by distinct references

```
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
```

• A[j], A[j+1] and A[j-1] have group reuse (spatial and temporal) in loop j
III. Reordering Transformations

• Analyze reuse in computation
• Apply loop reordering transformations to improve locality based on reuse
• With any loop reordering transformation, always ask
  - Safety? (doesn’t reverse dependences)
  - Profitability? (improves locality)
Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)
  for (j=0; j<6; j++)
```

```
for (j=0; j<6; j++)
  for (i= 0; i<3; i++)
```

Which one is better for row-major storage?
Safety of Permutation

- **Intuition:** Cannot permute two loops i and j in a loop nest if doing so reverses the direction of any dependence.

- Loops i through j of an n-deep loop nest are *fully permutable* if for all dependences D, either
  \[(d_1, \ldots, d_{i-1}) > 0\]
  or
  \[\text{forall } k, i \leq k \leq j, \ d_k \geq 0\]

- **Stated without proof:** Within the affine domain, n-1 inner loops of n-deep loop nest can be transformed to be fully permutable.
Simple Examples: 2-d Loop Nests

- Distance vectors
- Ok to permute?

for (i = 0; i<3; i++)
  for (j=0; j<6; j++)

for (i = 0; i<3; i++)
  for (j=0; j<6; j++)
Tiling (Blocking): Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time
Tiling Example

for (j=1; j<M; j++)
  for (i=1; i<N; i++)
    D[i] = D[i] + B[j,i]

Strip mine

for (j=1; j<M; j++)
  for (i=1; i<N; i+=s)
    for (ii=i, min(i+s-1,N))
      D[i] = D[i] + B[j,i]

Permute

for (i=1; i<N; i++)
  for (j=1; j<M; j++)
    for (ii=i, min(i+s-1,N))
      D[i] = D[i] + B[j,i]
Legality of Tiling

• Tiling = strip-mine and permutation
  - Strip-mine does not reorder iterations
  - Permutation must be legal
  OR
  - strip size less than dependence distance
IV. Locality Optimization

• Reuse analysis can be formulated in a manner similar to dependence analysis
  - Particularly true for temporal reuse
  - Spatial reuse requires special handling of most quickly varying dimension

• Simplification for today’s lecture
  - Estimate cache misses for different scenarios
  - Select scenario that minimizes misses
### Reuse Analysis:
Use to Estimate Cache Misses

For the given loops:

1. **Reference Loop**
   ```
   for (i=0; i<N; i++)
       for (j=0; j<M; j++)
   ```

2. **Loop J**
   ```
   for (j=0; j<M; j++)
       for (i=0; i<N; i++)
   ```

#### Reuse Analysis Table

<table>
<thead>
<tr>
<th>Reference</th>
<th>Loop J</th>
<th>Loop I</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>B[j,i]</td>
<td>M</td>
<td>N*M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
<th>Loop I</th>
<th>Loop J</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>N/cls(*)</td>
<td>M*N/cls</td>
</tr>
<tr>
<td>B[j,i]</td>
<td>N/cls</td>
<td>M*N/cls</td>
</tr>
</tbody>
</table>

(*) cls = Cache Line Size (in elements)
Allen & Kennedy: 
Innermost memory cost

- Innermost memory cost: $C_M(L_i)$
  - assume $L_i$ is innermost loop
    - $l_i$ = loop variable, $N$ = number of iterations of $L_i$
  - for each array reference $r$ in loop nest:
    - $r$ does not depend on $l_i$: cost ($r$) = 1
    - $r$ such that $l_i$ strides over a non-contiguous dimension:
      cost ($r$) = $N$
    - $r$ such that $l_i$ strides over a contiguous dimension:
      cost ($r$) = $N/c_l$s
  - $C_M(L_i) = \text{sum of cost} (r)$

Implicit in this cost function is that $N$ is sufficiently large that cache capacity is exceeded by data footprint in innermost loop
Canonical Example: matrix multiply
Selecting Loop Order

DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    END DO
  END DO
END DO

- \( C_M(I) = \frac{2N^3}{cl} + N^2 \)
- \( C_M(J) = 2N^3 + N^2 \)
- \( C_M(K) = N^3 + \frac{N^3}{cl} + N^2 \)
- Ordering by innermost loop cost: (J, K, I)
Canonical Example: Matrix Multiply
Selecting Tile Size

Choose $T_i$ and $T_k$ such that data footprint does not exceed cache capacity

```
DO K = 1, N  by T_k
  DO I = 1, N by T_i
    DO J = 1, N
      DO KK = K, min(KK+ T_k,N)
        DO II = I, min(II+ T_i,N)
          C(II,J)= C(II,J)+A(II,KK)*B(KK,J)
```

```
B_I

B_K

C

A

B
```
How to select optimal tile size? (topic for next week)

Square Tile with Lowest L1 Misses

Slide source: Jacqueline Chame
Next Week

• How to use loop reordering transformations in an auto-tuning optimization system?