Lecture 11
Pipelined Parallelism

1. Fully permutable loop nests & pipelining
2. Example: Transforming for full permutability
3. Time Affine Partitioning: Problem
4. Time Affine Partitioning Algorithm
5. \(O(1)\) Synchronization problem

Readings: Chapter 11.8-11.9

1. Maximum Parallelism & No Communication

C: Space partitioning of Computation to Processor ID
For every pair of data dependent accesses \(F_{1i_1}+f_1\) and \(F_{2i_2}+f_2\)

Find \(C_1, c_1, C_2, c_2\):

\[
\forall i_1, i_2 \quad F_{1i_1} + f_1 = F_{2i_2} + f_2 \rightarrow C_{1i_1} + c_1 = C_{2i_2} + c_2
\]

with the objective of maximizing the rank of \(C_1, C_2\)
**SOR (Successive Over-Relaxation): An Example**

for i = 1 TO m  
for j = 1 to n  

**Pipelineable Parallelism**

for i = 1 TO m  
for j = 1 to n  

Processor ID: p  
Synchronization variable: t[p] initialized to 0  
WAIT: thread waits until the condition becomes true

for j = 1 to n  
if (p==1) or (WAIT(t[p-1]>=j))  
t[p]++;

A Fully Permutable Loop Nest: Example

\[
\text{for } i = 1 \text{ TO } m \\
\text{for } j = 1 \text{ to } n \\
\]

\[
\text{for } j = 1 \text{ TO } n \\
\text{for } i = 1 \text{ to } m \\
\]

A Fully Permutable Loop Nest: Definition

- A loop nest is fully permutable if all the loops can be permuted arbitrarily without changing the semantics of the program.
**r-Dimensional Pipelineable Parallelism**

- r-deep fully permutable loop nest; \( r > 1 \)
  - \( r \) choices of outermost loops
  - \( r-1 \) degrees of parallelism
  - \( O(n^{r-1}) \) parallelism
  - \( O(n) \) synchronization

- Code generation
  - \( r-1 \) outer loops: processor ID (\( p_1, p_2, \ldots, p_{r-1} \))
  - Sequential \( r \)th loop: \( i_r \)
  - iteration \( i_r \) for processor \( (p_1, p_2, \ldots, p_{r-1}) \), waits for iteration \( i_r \) for processors \( (p_{1-1}, p_2, \ldots, p_{r-1}) \), \( (p_1, p_2^{-1}, \ldots, p_{r-1}) \), \ldots, \( (p_1, p_2, \ldots, p_{r-1-1}) \).

---

**Sequential Code**

- A sequential loop nest

  ![Diagram of sequential code with loops](image)

- Degree of the outermost permutable loop nest?

- Intuition: Fully permutable loop nest:
  - Dependence does not point backwards along any axis
Pipelined Parallelism

1. Fully permutable loop nests & pipelining
2. Example: Transforming for full permutability
3. Time Affine Partitioning: Problem
4. Time Affine Partitioning Algorithm
5. \( O(1) \) Synchronization problem

2. Example

\[
\begin{align*}
\text{for } i &= 0 \text{ to } m \\
\text{for } j &= 0 \text{ to } n \\
X[i+1] &= (X[j] + X[j+1] + X[j+2])
\end{align*}
\]
Transforming for Fully Permutable Loop Nests

for $i = 0$ TO $m$
for $j = 0$ to $n$

\[ X[i+1] = (X[j] + X[j+1] + X[j+2]) \]

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

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

\[ j' = i + j \]
\[ j = j' - i' \]

\[ i' = i \]
\[ 0 <= i' <= m \]
\[ j' = i + j \]
\[ 0 <= j' - i' <= n \]

Loop bounds:

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

for $i' = 0$ TO $m$
for $j' = i'$ to $i' + n$

\[ X[j' - i' + 1] = (X[j'] + X[j' + 1] + X[j' + 2]) / 3 \]

\[ X[j' - i' + 1] = (X[j' - i'] + X[j' - i' + 1] + X[j' - i' + 2]) / 3 \]
Pipelined Parallelism

1. Fully permutable loop nests & pipelining
2. Example: Transforming for full permutability
3. Time Affine Partitioning: Problem
4. Time Affine Partitioning Algorithm
5. $O(1)$ Synchronization problem
3. Recall: Pipelineable Parallelism

- r-deep fully permutable loop nest; r > 1
  - r choices of outermost loops
  - r-1 degrees of parallelism
  - \(O(n^{-1})\) parallelism
  - \(O(n)\) synchronization

- Goal: Find transformation to maximize the degree of pipelining
- \(\rightarrow\) Find all the possible outermost loops

---

**Finding the Maximum Degree of Pipelining**

C: Time partitioning of Computation to Time

For every pair of data dependent accesses \(F_{i_1} + f_1\) and \(F_{i_2} + f_2\)

Let \(B_1 i_1 + b_1 \geq 0\), \(B_2 i_2 + b_2 \geq 0\) be the corresponding loop bound constraints,

Find \(C_{i_1}, c_{i_1}, C_{i_2}, c_{i_2}\):

\[
\forall i_1, i_2 \quad B_1 i_1 + b_1 \geq 0, \quad B_2 i_2 + b_2 \geq 0 \\
(i_1 \leq i_2) \land (F_{i_1} + f_1 = F_{i_2} + f_2) \quad \rightarrow \quad C_{i_1} + c_1 \leq C_{i_2} + c_2
\]

with the objective of maximizing the rank of \(C_{i_1}, C_{i_2}\)
### Solutions of Time Mapping

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

### Solutions to Loop Transforms

\[
\begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix}
\]
Pipelined Parallelism

1. Fully permutable loop nests & pipelining
2. Example: Transforming for full permutability
3. Time Affine Partitioning: Problem
4. Time Affine Partitioning Algorithm
5. \(O(1)\) Synchronization problem

4. Communication-Free vs Pipelining

Loops

\[ \begin{align*}
F_{1i_1} + f_1 & \\
F_{2i_2} + f_2 & \\
C_{2i_2} + c_2 & \\
C_{1i_1} + c_1 & \\
\end{align*} \]

Array

Processor ID

\[ \overset{\text{Time Stage}}{i_1 \leq i_2} \]

\[ \overset{\text{Loops}}{\begin{align*}
F_{1i_1} + f_1 & \\
F_{2i_2} + f_2 & \\
C_{2i_2} + c_2 & \\
C_{1i_1} + c_1 & \\
\end{align*}} \]

Array
Comparing the Two Problems

Communication-Free Parallelism:
C: Space partitioning of Computation to Processor ID
For every pair of data dependent accesses \( F_i + f_1 \) and \( F_j + f_2 \)

Find \( C_{i}, c_{1}, C_{j}, c_{2} \):

\[ \forall i, j \quad F_i + f_1 = F_j + f_2 \Rightarrow C_i + c_1 = C_j + c_2 \]

with the objective of maximizing the rank of \( C_{i}, C_{j} \)

Pipelining Parallelism:
C: Time mapping of Computation to Time
For every pair of data dependent accesses \( F_i + f_1 \) and \( F_j + f_2 \)

Let \( B_{i} + b_{1} \geq 0, B_{j} + b_{2} \geq 0 \) be the corresponding loop bound constraints,

Find \( C_{i}, c_{1}, C_{j}, c_{2} \):

\[ \forall i, j \quad B_{i} + b_{1} \geq 0, B_{j} + b_{2} \geq 0, (i \leq j) \rightarrow C_i + c_1 \leq C_j + c_2 \]

with the objective of maximizing the rank of \( C_{i}, C_{j} \)

Farkas Lemma

Finding the possible time dimensions \( c \):

Given matrix \( A \), find a vector \( c \) such that

for all vectors \( x \) such that \( Ax \geq 0 \),

\[ c^{\top}x \geq 0 \]

Farkas Lemma, 1901 (real domain)
The primal system of inequalities

\[ Ax \geq 0, \quad c^{\top}x < 0 \]

has a real-valued solution \( x \)
or, the dual system

\[ A^{\top}y = c, \quad y \geq 0 \]

has a real-valued solution \( y \), but never both.

Time partitioning: Find \( c \) such that \( A^{\top}y = c, \quad y \geq 0 \)

Note: Farkas Lemma: a theorem of the alternative
(no intuitive proof exists)
Cholesky Decomposition

for (i = 1; i <= N; i++) {
    for (j = 1; j <= i-1; j++) {
        for (k = 1; k <= j-1; k++) {
            X[i,j] = X[i,j] - X[i,k]*X[j,k];
        }
    }
    for (m=1; m<=i-1; m++) {
        if (j==i && j<i) {
            X[i,j] = X[i,j]/X[j,j];
        }
    }
    X[i,i] = sqrt(X[i,i]);
}

Blocking with Matrix Multiplication

• Original program
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
        Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
      }
    }
  }

• Stripmine 2 outer loops
  for (ii = 0; ii < n; ii = ii+8) {
    for (i = ii; i < min(n,ii+8); i++) {
      for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
        }
      }
    }
  }

  • Permute loops
    for (ii = 0; ii < n; ii = ii+8) {
      for (jj = 0; jj < n; jj = jj+8) {
        for (i = ii; i < min(n,ii+8); i++) {
          for (j = jj; j < min(n,jj+8); j++) {
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
          }
        }
      }
    }

Blocking

- Fully permutable loop nests can be blocked
  - Stripmine to create more fully permutable loops
  - Permutable loops can be moved inside (by definition)
- Uses
  - Increase data locality
    - Block size can be chosen so data accessed in the block fits in memory
  - Reduce synchronization overhead
    - By a factor of the block size
    - Consideration: startup latency, load balance for triangular loops

Pipelined Parallelism

1. Fully permutable loop nests & pipelining
2. Example: Transforming for full permutability
3. Time Affine Partitioning: Problem
4. Time Affine Partitioning Algorithm
5. \(O(1)\) Synchronization problem
5. Beyond Pipelined Parallelism

for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];     (s1)
    W[A[i]] = X[i];        (s2)
}

O(1) Synchronization

for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];     (s1)
    W[A[i]] = X[i];        (s2)
}

- Program dependence graph
  - Nodes: statements
  - Edges: data dependence

- Split the program into
  a sequence of strongly connected components
  separated by barriers
**Algorithm**

1. Find parallelism with minimum synchronization
   - Find outermost communication-free parallelism
   - Find outermost fully permutable loop nest
   - If there are inner loops remaining
     - Find program dependence graph
     - Split the program into strongly connected components
     - Repeat for each strongly connected component

2. Blocking can be applied based on machine characteristics

---

**Summary: Two Key Algorithms**

![Diagram showing loops and arrays with processor ID and time stage]

- Loops and arrays with processor ID and time stage relationships.
- Key stages and transformations indicated.

---

M. Lam  
CS243: Loop Transformations  
29
Example: Neural Network

// 2D 3x3 convolution (stride=1)
for i = 0 to channels-1
for y = 2 to Sy-1
for x = 2 to Sx-1
B[i,y,x] = A[i,y-2,x-2]*W1[0,0] + A[i,y-2,x-1]*W1[0,1] + ...
A[i,y-1,x-2]*W1[1,0] + ...
A[i,y-1,x-1]*W1[1,1] + ...
A[i,y,x-2]*W1[2,0] + ...
A[i,y,x-1]*W1[2,1] + ...

// ReLU (Rectified Linear Unit)
for i = 0 to channels-1
for y = 2 to Sy-1
for x = 2 to Sx-1
B[i,y,x] = max(B[i,y,x], 0)

// 2D convolution (Stride = 2)
if (y >=4) && (x >=4) && (y mod 2 == 0) && (x mod 2 == 0)
C[i,y/2,x/2] += B[i,y-2,x-2]*W2[0,0] + ...
B[i,y-1,x-2]*W2[1,0] + ...

// Dense neural network layer
for j = 0 to Sj-1
for y = 2 to (Sy-1)/2
for x = 2 to (Sx-1)/2
D[i,j] = B[i,y,x]*W3[j,y,x]

// Softmax
for j = 0 to Sj-1
T[i,j] = exp(D[i,j])
E[i] += T[i,j]
for j = 0 to Sj-1
F[i,j] = exp(T[i,j])/E[i]

Parallelization without Reduction Optimization

// 2D convolution (stride=1)
for i = 0 to channels-1
for y = 2 to Sy-1
for x = 2 to Sx-1
B[i,y,x] = A[i,y-2,x-2]*W1[0,0] + A[i,y-2,x-1]*W1[0,1] + ...
A[i,y-1,x-2]*W1[1,0] + ...
A[i,y-1,x-1]*W1[1,1] + ...
A[i,y,x-2]*W1[2,0] + ...
A[i,y,x-1]*W1[2,1] + ...

// ReLU (Rectified Linear Unit)
for i = 0 to channels-1
for y = 2 to Sy-1
for x = 2 to Sx-1
B[i,y,x] = max(B[i,y,x], 0)

// 2D convolution (Stride = 2)
if (y >=4) && (x >=4) && (y mod 2 == 0) && (x mod 2 == 0)
C[i,y,x] = B[i,y-2,x-2]*W2[0,0] + ...
B[i,y-1,x-2]*W2[1,0] + ...

// Dense neural network layer
for j = 0 to Sj-1
for y = 2 to (Sy-1)/2
for x = 2 to (Sx-1)/2
D[i,j] = B[i,y,x]*W3[j,y,x]

// Softmax
for j = 0 to Sj-1
T[i,j] = exp(D[i,j])
E[i] += T[i,j]
for j = 0 to Sj-1
F[i,j] = exp(T[i,j])/E[i]