Lecture 11
Pipelined Parallelism

1. Fully permutable loop nests, pipelining, blocking
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. Recall: 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 $C_1, c_1, C_2, c_2$: 

$$\forall \ i_1, i_2 \quad F_1 i_1 + f_1 = F_2 i_2 + f_2 \rightarrow C_1 i_1 + c_1 = C_2 i_2 + c_2$$

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

\[
\text{for } i = 1 \text{ TO } m \\
\quad \text{for } j = 1 \text{ to } n \\
\quad \quad A[i,j] = c \times (A[i-1,j] + A[i,j-1])
\]
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]++;

Fully Permutable Loop Nests

- **Definition:**
  A loop nest is fully permutable if all the loops can be permuted arbitrarily without changing the semantics of the program.

- **Example:**

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

  The matrix transformation:
  
  $$
  \begin{bmatrix}
  j' \\
  i'
  \end{bmatrix} = \begin{bmatrix}
  0 & 1 \\
  1 & 0
  \end{bmatrix} \begin{bmatrix}
  i \\
  j
  \end{bmatrix}
  $$

  ```plaintext
  for j = 1 TO n
      for i = 1 to m
  ```
When is a Loop Fully Permutable?

• Sequential execution order:

• A loop nest is fully permutable if all the dependences are satisfied in the sequential execution order, under all loop permutations

• INTUITION: A loop nest is fully permutable if
  – Its dependences do not point backwards along any axis

• Relationship between communication-free parallelism & full permutability?
r-Dimensional Pipelineable Parallelism

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

- Synchronization
  - \( r-1 \) 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 \((p_1-1, p_2, \ldots, p_{r-1})\),
    \((p_1, p_2-1, \ldots, p_{r-1}), \ldots,
    (p_1, p_2, \ldots, p_{r-1}-1)\).
Recall: Blocking for Matrix Multiplication

\[
\begin{align*}
\begin{array}{c}
\text{Data} \\
\text{Accessed}
\end{array}
\end{align*}
\]

- 32 \times 1000 = 32,000
- 1000 \times 1000 = 1,000,000
- 1000 \times 32 = 32,000

- Total Data Accessed: 1002000
- Total Data Accessed: 65024
Experimental Results

With Blocking
Without Blocking

Speedup

Processors
How to Block Loops?

- Fully permutable loop nests can be blocked
  1. Stripmine to create more fully permutable loops
     for (i = 0; i < n; i++) {
       <code>
     }
  =>
     for (ii = 0; ii < n; ii = ii+B) {
       for (i = ii; i < min(n,ii+B); i++) {
         <code>
       }   
     }
  2. Permute inner stripmined loop inside
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+B) {
      for (i = ii; i < min(n,ii+B); i++) {
          for (jj = 0; jj < n; jj = jj+B) {
              for (j = jj; j < min(n,jj+B); j++) {
                  Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
              }
          }
      }
  }
  ```

- **Permute loops**
  ```
  for (ii = 0; ii < n; ii = ii+B) {
      for (jj = 0; jj < n; jj = jj+B) {
          for (k = 0; k < n; k++) {
              for (i = ii; i < min(n,ii+B); i++) {
                  for (j = jj; j < min(n,jj+B); j++) {
                      Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
                  }
              }
          }
      }
  }
  ```
Uses of Blocking

• Increase data locality
  – Block size can be chosen
    so data accessed in the block fits in the faster hierarchy
      (virtual memory, cache, registers)
• Reduce synchronization overhead
  – By a factor of the block size
  – Consideration: startup latency, load balance for triangular loops
• SIMD instructions
  – To create contiguous vector access
    \[
    \begin{align*}
    &\text{for } (j = 0; j < n; j++) \{ \\
    &\quad \text{for } (k = 0; k < n; k++) \{ \\
    &\quad\quad Z[i,j] = Z[i,j] + X[i,k]*Y[k,j]; \\
    &\quad\}\}
    \end{align*}
    \]
    => \[
    \begin{align*}
    &\text{for } (jj = 0; jj < n; jj+=4) \{ \\
    &\quad \text{for } (k = 0; k < n; k++) \{ \\
    &\quad\quad \text{for } (j = jj; jj < \min(n, jj+4); j++) \{ \\
    &\quad\quad\quad Z[i,j] = Z[i,j] + X[i,k]*Y[k,j]; \\
    &\quad\quad\}\}
    &\quad\}\}
    \end{align*}
    \]
2. How to Make Loops Fully Permutable? (example)

Example:

```
for i = 0 TO m
    for j = 0 to n
        X[j+1] = (X[j] + X[j+1] + X[j+2])
```

![Diagram 1]

![Diagram 2]
Transforming for Full Permutability

for \( i = 0 \) TO \( m \)
for \( j = 0 \) to \( n \)
\[ X[j+1] = (X[j] + X[j+1] + X[j+2]) \]
Code Generation

for $i = 0$ TO $m$
for $j = 0$ to $n$
$X[j+1]=(X[j]+X[j+1]+X[j+2])/3$

for $i' = 0$ TO $m$
for $j' = i'$ to $i'+n$
$X[j'-i'+1]=(X[j'-i']]+X[j'-i'+1]+X[j'-i'+2])/3$

Loop bounds:

$i' = i$ \hspace{1cm} 0 <= $i'$ <= $m$

$j' = i + j$ \hspace{1cm} 0 <= $j'$- $i'$ <= $n$

$j = j' - i'$. 

\[
\begin{bmatrix}
  i' \\
  j'
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 \\
  1 & 1
\end{bmatrix}
\begin{bmatrix}
  i \\
  j
\end{bmatrix}
\]
Is the Result Fully Permutable?

\[
\begin{bmatrix}
  i' \\ j'
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 \\
  1 & 1
\end{bmatrix}
\begin{bmatrix}
  i \\ j
\end{bmatrix}
\]
3. The Problem of Creating Fully Permutable Loops

- **RECALL**: 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

- **GOAL**: Find transformation to maximize the degree of pipelining
- → 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 \( F_{1i1} + f_1 \) and \( F_{2i2} + f_2 \)
Let \( B_{1i1} + b_1 \geq 0, \ B_{2i2} + b_2 \geq 0 \) be the corresponding loop bound constraints,
Find \( C_1, \ c_1, \ C_2, \ c_2 \):
\[
\forall \ i_1, \ i_2 \quad B_{1i1} + b_1 \geq 0, \ B_{2i2} + b_2 \geq 0
\]
\[
(i_1 \leq i_2) \land (F_{1i1} + f_1 = F_{2i2} + f_2) \rightarrow C_{1i1} + c_1 \leq C_{2i2} + c_2
\]
with the objective of maximizing the rank of \( C_1, \ C_2 \)
Solutions of Time Mapping

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

\[
\begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix}
\]
Solutions to Loop Transforms

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

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

4. Time Partitioning Algorithm

Loops

\[ F_{i_{1}} + f_{1} \]
\[ F_{i_{2}} + f_{2} \]
\[ C_{i_{1}} + c_{1} \]
\[ C_{i_{2}} + c_{2} \]

Processor ID

Array

\[ i_{1} \leq i_{2} \]

Loops

\[ F_{i_{2}} + f_{2} \]
\[ F_{i_{1}} + f_{1} \]
\[ C_{i_{1}} + c_{1} \]
\[ C_{i_{2}} + c_{2} \]

Time Stage
Comparing the Two Problems

Communication-Free Parallelism:

C: Space partitioning of Computation to Processor ID
For every pair of data dependent accesses $F_1 i_1 + f_1$ and $F_2 i_2 + f_2$

Find $C_1, c_1, C_2, c_2$:

$$\forall i_1, i_2 \quad F_1 i_1 + f_1 = F_2 i_2 + f_2 \rightarrow C_1 i_1 + c_1 = C_2 i_2 + c_2$$

with the objective of maximizing the rank of $C_1, C_2$

Pipelining Parallelism:

C: Time mapping of Computation to Time
For every pair of data dependent accesses $F_1 i_1 + f_1$ and $F_2 i_2 + f_2$

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

Find $C_1, c_1, C_2, c_2$:

$$\forall i_1, i_2 \quad B_1 i_1 + b_1 \geq 0, B_2 i_2 + b_2 \geq 0$$
$$\quad (i_1 \leq i_2) \land (F_1 i_1 + f_1 = F_2 i_2 + f_2) \rightarrow C_1 i_1 + c_1 \leq C_2 i_2 + c_2$$

with the objective of maximizing the rank of $C_1, C_2$
Farkas Lemma

Finding the possible time dimensions \( c \):
Given matrix \( A \), find a vector \( c \) such that
for all vectors \( x \) such that \( Ax \geq 0 \),
\( c^T x \geq 0 \)

Farkas Lemma, 1901 (real domain)
The primal system of inequalities
\( Ax \geq 0, \; c^T x < 0 \)
has a real-valued solution \( x \)
or, the dual system
\( A^T y = c, \; y \geq 0 \)
has a real-valued solution \( y \), but never both.

Time partitioning: Find \( c \) such that \( A^T y = c, \; y \geq 0 \)

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

```plaintext
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];
        X[i,j] = X[i,j]/X[j,j];
    }
    for (m=1; m<=i-1; m++) {
        X[i,i]=X[i,i]-X[i,m]*X[i,m];
    }
    X[i,i] = sqrt(X[i,i]);
}
```

Transformed Space

```plaintext
for (i = 1; i <= N; i++) {
    for (j = 1; j <= i; j++) {
        for (k = 1; k <= i; k++)
            if (j<i && k<j)
                X[i,j] = X[i,j] - X[i,k]*X[j,k];
            if (j==k && j<i)
                X[i,j] = X[i,j]/X[j,j];
            if (i==j && k<i)
                X[i,i]=X[i,i]-X[i,k]*X[i,k];
            if (i==j && j==k)
                X[i,i] = sqrt(X[i,i]);
    }
}
```
5. O(1) Synchronization

What if there is only 1 fully permutable outermost loop?

Example:

```c
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];  (s1)
    W[A[i]] = X[i];      (s2)
}
```
**O(1) Synchronization Algorithm**

```plaintext
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 $O(1)$ barriers

```plaintext
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];      (s1)
}
```
Coarsest Parallelism with Minimum Synchronization

1. Find parallelism with coarsest parallelism with minimum synchronization
   
   a. Find outermost communication-free parallelism
   b. Find O(1) synch. parallelism to partitions found.
      (if no component has communication-free parallelism, leave as one partition).
   c. For each partition found,
      Find outermost fully permutable loop nest O(n) synch
      Find O(1) synch. parallelism on each processor partition, or on the sequential loop body otherwise.
   d. Recursively apply c to inner loops if any.

2. Apply blocking to improve locality
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-2]*W1[2,0] + ...

// 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 3x3 convolution (Stride = 2)
for i = 0 to channels-1
  for y = 2 to (Sy-1)/2
    for x = 2 to (Sx-1)/2
      C[i,y,x] = B[i,2*y-2,2*x-2]*W2[0,0] + ...
      B[i,2*y-1,2*x-2]*W2[1,0] + ...

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

// Softmax:
for i = 0 to channels-1
  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] = T[i,j]/E[i]
Parallelization without Reduction Optimization

// 2D convolution (stride=1)
for i = 0 to channels-1 // Parallel loop
    for y = 2 to Sy-1 // Permutable loop nest
        for x = 2 to Sx-1 // Permutable loop nest
            // 2D convolution
            B[i,y,x] += A[i,y-2,x-2]*W[0,0] + A[i,y-2,x-1]*W[0,1] + ...
            A[i,y-1,x-2]*W[1,0] + ...
            A[i,y-2,x]*W[2,0] + ...
            // ReLU (Rectified Linear Unit)
            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]*W[0,0] + ...
    B[i,y-1,x-2]*W[1,0] + ...

// Dense neural network layer
for j = 0 to Sj-1 /* Parallel loop */
    for y = 2 to (Sy-1)/2
        for x = 2 to (Sx-1)/2
            D[i,j] += C[i,y,x]*W3[j,y,x]
            T[i,j] = exp(D[i,j]);

// Softmax
for j = 0 to Sj-1
    E[i] += T[i,j];
for j = 0 to Sj-1 /* Parallel loop */
    F[i,j] = T[i,j]/E[i]
Summary: Two Key Algorithms

Loops

Processor ID

Array

\[ F_1i_1 + f_1 \]
\[ F_2i_2 + f_2 \]
\[ C_1i_1 + c_1 \]
\[ C_2i_2 + c_2 \]

\( i_1 \leq i_2 \)

Loops

Array

\[ F_1i_1 + f_1 \]
\[ F_2i_2 + f_2 \]
\[ C_1i_1 + c_1 \]
\[ C_2i_2 + c_2 \]

Time Stage