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 $F_{i_1}+f_1$ and $F_{i_2}+f_2$

Find $C_1$, $c_1$, $C_2$, $c_2$:

$\forall i_1, i_2 \quad F_{i_1}+f_1 = F_{i_2}+f_2 \Rightarrow C_{i_1}+c_1 = C_{i_2}+c_2$

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
      \[ A[i,j] = c \times (A[i-1,j] + A[i,j-1]) \]

**Pipelineable Parallelism**

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

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]++;

M. Lam  CS243: Pipelined Parallelism
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

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

When is a Loop Fully Permutable?

- **Sequential execution order:**

```
  i
  j
```

- 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⁻¹) parallelism
  - O(n) synchronization

- Synchronization
  - r-1 outer loops: processor ID (p₁, p₂, ..., pᵣ⁻¹)
  - Sequential rth loop: iᵣ
  - iteration iᵣ for processor (p₁, p₂, ..., pᵣ⁻¹), waits for iteration iᵣ for processors (p₁⁻¹, p₂⁻¹, ..., pᵣ⁻¹), (p₁, p₂⁻¹, ..., pᵣ⁻¹), ..., (p₁, p₂, ..., pᵣ⁻¹⁻¹).

---

**Recall: Blocking for Matrix Multiplication**

\[
\begin{align*}
\begin{array}{c}
\text{1000} \\
\text{32}
\end{array}
\end{align*}
\begin{align*}
\begin{array}{c}
\text{1000} \\
\text{1000}
\end{array} &= \begin{array}{c}
X
\end{array} \quad \text{Data Accessed} = 1002000
\end{align*}
\begin{align*}
\begin{array}{c}
\text{1000} \\
\text{32}
\end{array} &= \begin{array}{c}
X
\end{array} \quad \text{Data Accessed} = 65024
\end{align*}
\]
Experimental Results

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++) {
     \texttt{<code>}
     } =>
     for (ii = 0; ii < n; ii = ii+B) {
     for (i = ii; i < \text{min}(n,ii+B); i++) {
     \texttt{<code>}
     }
     }
     ```
  2. Permute inner stripmined loop inside
Blocking with Matrix Multiplication

- **Original program**
  ```c
  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**
  ```c
  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++) {
          for (k = 0; k < n; k++) {
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
          }
        }
      }
    }
  }
  ```
- **Permute loops**
  ```c
  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
  ```c
  for (jj = 0; jj < n; jj+=4) {
    for (k = 0; k < n; k++) {
      for (j = jj; j < min(n, jj+4); j++) {
        Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
      }
    }
  }
  ```
  ```c
  => for (jj = 0; jj < n; jj+=4) {
    for (k = 0; k < n; k++) {
      for (j = jj; j < min(n, jj+4); j++) {
        Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
      }
    }
  }
  ```
2. How to Make Loops Fully Permutable? (example)

Example:

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

Transforming for Full Permutability

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

\[
\text{for } i = 0 \text{ TO } m \\
\text{for } j = 0 \text{ to } n \\
X[j+1] = (X[j] + X[j+1] + X[j+2]) / 3
\]

Loop bounds:

- \( i' = i \) \quad 0 \leq i' \leq m
- \( j' = i + j \) \quad 0 \leq j'-i' \leq n
- \( j'' = i + j'' \)

**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
  - \( \rightarrow \) 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 \( F_{i1}+f_1 \) and \( F_{i2}+f_2 \)

Let \( B_{i1} + b_1 \geq 0 \), \( B_{i2} + b_2 \geq 0 \) be the corresponding loop bound constraints,

Find \( C_{i1}, c_1, C_{i2}, c_2 \):

\[
\forall i_1, i_2 \quad \begin{align*}
B_{i1} + b_1 & \geq 0, \quad B_{i2} + b_2 \geq 0 \\
(i_1 \leq i_2) \land (F_{i1}+f_1 = F_{i2}+f_2) & \rightarrow C_{i1} + c_1 \leq C_{i2} + c_2
\end{align*}
\]

with the objective of maximizing the rank of \( C_{i1}, C_{i2} \)
Solutions of Time Mapping

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

Solutions to Loop Transforms

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

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

\[
\begin{bmatrix}
i \\
j
\end{bmatrix}
\]

\[
\begin{bmatrix}
i' \\
j'
\end{bmatrix}
\]
4. Time Partitioning Algorithm

Comparing the Two Problems

Communication-Free Parallelism:
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 $C_{i_1}, c_1, C_{i_2}, c_2$:
$\forall i_1, i_2 \, F_{1i_1} + f_1 = F_{2i_2} + f_2 \Rightarrow C_{i_1} + c_1 = C_{i_2} + c_2$
with the objective of maximizing the rank of $C_{i_1}, C_{i_2}$

Pipelining Parallelism:
C: Time mapping of Computation to Time
For every pair of data dependent accesses $F_{1i_1} + f_1$ and $F_{2i_2} + f_2$
Let $B_{1i_1} + b_1 \geq 0$, $B_{2i_2} + b_2 \geq 0$ be the corresponding loop bound constraints,
Find $C_{i_1}, c_1, C_{i_2}, c_2$:
$\forall i_1, i_2 \, B_{1i_1} + b_1 \geq 0, \, B_{2i_2} + b_2 \geq 0$
$(i_1 \leq i_2) \land (F_{1i_1} + f_1 = F_{2i_2} + f_2) \Rightarrow C_{i_1} + c_1 \leq C_{i_2} + c_2$
with the objective of maximizing the rank of $C_{i_1}, C_{i_2}$
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

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

Transformed Space

```plaintext
i = 6
```

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

What if there is only 1 fully permutable outermost loop?

Example:
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
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
      \[ B[i,y,x] = A[i,y-2,x-2]*W1[0,0] + A[i,y-2,x-1]*W1[0,1] + ... \\
      A[i,y-1,x-2]*W1[1,0] + ... \]
      \[ A[i,y,x-2]*W1[2,0] + ... \]

// 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
      \[ C[i,y,x] = B[i,y-2*2,x-2*2]*W2[0,0] + ... \]
      \[ B[i,2*2-1,x-2*2]*W2[1,0] + ... \]

// 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
        \[ B[i,j] = C[i,y,x]*W3[i,j,x] \]

// ReLU
for i = 0 to channels-1
  for j = 0 to Sj-1
    for y = 2 to Sy-1
      for x = 2 to Sx-1
        \[ B[i,j] = \max(B[i,j], 0) \]

// Softmax
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
        \[ T[i,j] = \exp(D[i,j]) \]
        \[ E[i] = T[i,j] \]
        \[ F[i,j] = T[i,j]/E[i] \]

// Softmax
for i = 0 to channels-1
  for j = 0 to Sj-1
    for y = 2 to Sy-1
      for x = 2 to Sx-1
        \[ T[i,j] = \exp(D[i,j]) \]
        \[ E[i] = T[i,j] \]
        \[ F[i,j] = T[i,j]/E[i] \]
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]*W1[0,0] + A[i,y-2,x-1]*W1[0,1] + ...
      A[i,y-1,x-2]*W1[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]*W2[0,0] + ...
  B[i,y-1,x-2]*W2[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

Carnegie Mellon

CS243: Pipelined Parallelism