Lecture 11
Pipelined Parallelism

1. Intuition: Time mapping
2. Affine Time Partitioning Problem
3. Affine Time Partitioning Algorithm
4. Coarsest-Grain Parallelization Algorithm
5. Blocking
6. Real-world examples

Readings: Chapter 11.8-11.9
Lecture 10: 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 $C_1$, $C_2$
1. SOR (Successive Over-Relaxation): An Example

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

Quiz:
- Is there communication-free parallelism?
- Can you find parallelism with communication?
**SOR (Successive Over-Relaxation): An Example**

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

Focusing on sequential execution

- i is a legal outer loop
- Is j also a legal outer loop?
- Two independent basis vectors for the outer loop:
  \[[1 \ 0], [0 \ 1]\]
- Any combination of \[1 \ 0\] and \[0 \ 1\] is a legal (DoAll) outer loop!
- Example: \[1 \ 1\]
SOR (Successive Over-Relaxation): An Example

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

Focusing on sequential execution

- \( i \) is a legal outer loop
- Is \( j \) also a legal outer loop?
- Two independent basis vectors for the outer loop:
  \[
  [1 \ 0], [0 \ 1]
  \]
- Any combination of \([1 \ 0], [0 \ 1]\) is a legal (DoAll) outer loop!
- Example: \([1 \ 1]\)
  - All original data dependences do not point backward in time
  - A wavefront of execution
**SOR (Successive Over-Relaxation): An Example**

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

Focusing on sequential execution
- Two independent basis vectors for the outer loop:
  \[
  \begin{bmatrix} 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \end{bmatrix}
  \]
- Any combination of \([1 \ 0], [0 \ 1]\) is a legal (DoAll) outer loop!
- Example: \([1 \ 2]\)
  - All original data dependences do not point backward in time
  - A wavefront of execution
General Observation

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

- Multiple legal outer loops:
  - Choice in execution $\rightarrow$ there is parallelism
- 2 independent basis vectors
  - The solution space has rank 2
  - 1 degree of pipelined parallelism
- $r$ independent basis vectors
  - The solution space has rank $r$
  - $r-1$ degrees of pipelined parallelism

Choice means parallelism
Implementing Wavefronts Without Barriers

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

Assign each row i to a processor
Processor ID: p = i
Synchronization variable: t[p] = 0 (iterations executed)
WAIT: thread waits until the condition becomes true

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

- Good locality
- Relaxed wavefront
- O(n) synchronization overhead
2. 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 $C_1$, $c_1$, $C_2$, $c_2$:

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

with the objective of maximizing the rank of $C_1$, $C_2$
Problem Statement: Maximum Pipelinable Parallelism

C: Time Partitioning of Computation to Time Step
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 $C_1$, $c_1$, $C_2$, $c_2$:
\[
\forall i_1, i_2 \quad B_1i_1 + b_1 \geq 0, \quad B_2i_2 + b_2 \geq 0 \\
(i_1 \leq i_2) \land (F_1i_1+f_1 = F_2i_2+f_2) \rightarrow C_1i_1+c_1 \leq C_2i_2+c_2
\]
with the objective of maximizing the rank of $C_1$, $C_2$
Example 1

- Solving the equations yield two independent basic vectors: $[1 \ 0], [0 \ 1]$
Example 1

2 time mappings:

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

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

→ 2 legal permutations

(a) \[
[t_1] = \begin{bmatrix} 1 & 0 \\ i & j \end{bmatrix}
\]

(b) \[
[t_1] = \begin{bmatrix} 0 & 1 \\ i & j \end{bmatrix}
\]
Example 2

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

Is the above loop permutable as is?
(Can we make both i and j outer loops?)
for $i = 0$ TO $m$
for $j = 0$ to $n$
$X[j+1] = (X[j] + X[j+1] + X[j+2]) / 3$

2 time mappings:

$[t] = [1, 0] \begin{bmatrix} i \\ j \end{bmatrix} + [0]$

$[t] = [1, 1] \begin{bmatrix} i \\ j \end{bmatrix} + [0]$

Intuitively:
The time mapping makes all the dependences not point backwards.
Time Partitioning Results

2 time mappings:

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

2 permutations:

\[
(a) \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

\[
(b) \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

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

Transformation

$i' = i$

$0 <= i' <= m$

$j' = i + j$

$0 <= j' - i' <= n$

Loop bounds (Using Fourier-Motzkin Elimination)
Fully Permutable Loop Nests

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

• Affine time partitioning algorithm finds fully permutable loop nests.

• Given rank $r$ time mappings:
  – There are $r$ independent legal outermost loops
  – Dependences do not point backward along $r$-axes
  – Rank $r$ matrix (comprising $r$ independent basis vectors)
    transforms the original loop
    into $r$-deep outermost fully permutable nest

• Generate $(r - 1)$ dimensional wavefront from permutable loop nest
  ($r - 1$ degrees of parallelism)
r-Dimensional Pipelineable Parallelism

- $r$-dimensions of legal time mapping:
  - $r$-1 degrees of parallelism
    - $O(n^{r-1})$ parallelism, $n$ is the number of iterations in each loop
    - $O(n)$ synchronization

- Synchronization
  - processor ID ($p_1, p_2, ..., p_{r-1}$):
    - $r$-1 outer loops map to each processor
    - Runs $r$th loop sequentially on each processor
  - iteration $i_r$ for processor ($p_1, p_2, ..., p_{r-1}$), waits for its $r$-1 neighbors
    - iteration $i_r$ for processors ($p_{1-1}, p_2, ..., p_{r-1}$),
      ($p_1, p_{2-1}, ..., p_{r-1}$), ..., ($p_1, p_2, ..., p_{r-1-1}$).
3. How to Compute Time Partitioning?

Compare:

Loops

Array

Processor ID

Time Stage

\[ F_{1i_1} + f_1 \]

\[ F_{2i_2} + f_2 \]

\[ C_{1i_1} + c_1 \]

\[ C_{2i_2} + c_2 \]
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 $C_1$, $c_1$, $C_2$, $c_2$:

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

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

**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 $C_1$, $c_1$, $C_2$, $c_2$:

$$\forall \ i_1, i_2 \quad B_{1i_1} + b_1 \geq 0, \ B_{2i_2} + b_2 \geq 0$$

$$(i_1 \leq i_2) \land (F_{1i_1} + f_1 = F_{2i_2} + f_2) \rightarrow C_{1i_1} + c_1 \leq C_{2i_2} + c_2$$

with the objective of maximizing the rank of $C_1$, $C_2$
Farkas Lemma Comes to the Rescue!

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
\[ Ax \geq 0, \quad c^T x < 0 \]
has a real-valued solution $x$
or, the dual system
\[ A^T y = c, \quad y \geq 0 \]
has a real-valued solution $y$, but never both.

Time partitioning: Find $c$ such that $A^T y = c, \quad y \geq 0$ \hspace{1cm} Much easier!

Note: Farkas Lemma: a theorem of the alternative (no intuitive proof exists)
4. Coarsest Granularity of Parallelism

What if there is only 1 legal outermost loop?

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

O(1) Synchronization Algorithm (single loop barrier)

```cpp
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

- **Algorithm**
  - Split the program into
    a sequence of strongly connected components
  - separated by O(1) barriers
  - Find communication-free parallelism in components
    (if no component has communication-free parallelism,
    leave as one partition).
Coarsest Parallelism with Minimum Synchronization

Find parallelism with coarsest parallelism with minimum synchronization

a. Find outermost communication-free parallelism
b. Find $O(1)$ sync. parallelism to split parallelizable code
c. For each parallelizable code unit found,
   Find outermost fully permutable loop nest $O(n)$ sync
   Find $O(1)$ sync. parallelism to split parallelizable code
d. Recursively apply c to inner loops if any.
5. Blocking

for (i = 1; i < m; i++) {
    for (j = 1; j < n; j++) {
    }
}

Reduces synchronization
Increases locality
minimize communication
improve cache performance
(applicable to registers as well!)
Stripmining & Blocking

- All loops can be stripmined

- Let B be the block size

  Stripmine a loop into 2

  ```
  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>
      }
  }
  ```

- If the original loop is permutable, stripmined loops are permutable

- Permute loops to create blocks
Blocking SOR

For simplicity, assume \( m = n = 101; B = 10 \)

- **Original program**
  ```
  for (i = 1; i < 101; i++) {
    for (j = 1; j < 101; j++) {
    }
  }
  ```

- **Stripmine loops**
  ```
  for (ii = 1; ii < 101; ii = ii+10) {
    for (i = ii; i < ii+10; i++) {
      for (jj = 1; jj < 101; jj = jj+10) {
        for (j = jj; j < jj+10; j++) {
        }
      }
    }
  }
  ```

- **Permute loops**
  ```
  for (ii = 1; ii < 101; ii = ii+10) {
    for (jj = 1; jj < 101; jj = jj+10) {
      for (i = ii; i < ii+10; i++) {
        for (j = jj; j < jj+10; j++) {
        }
      }
    }
  }
  ```
Another Use of Blocking

- Coarse-grain parallel loops can be blocked to improve parallelism or locality in inner loops.

  Example: multiprocessor, with instruction-level parallelism

  ```
  for (i = 0; i < n; i++) { //doall
    for (j = 1; j < n; j++) { //sequential loop
      Z[i, j] = Z[i, j-1]
    }
  }
  
  => for (ii = 0; ii < n; ii+=16) { //doall
    for (j = 0; j < n; j++) { //sequential loop
      for (i = ii; ii < min(n, ii+16); i++) { //doall
        Z[i, j] = Z[i, j-1];
      }
    }
  }
  ```
6. Examples

- Real programs can be parallelized with affine transforms
  - Machine learning: convolutional neural networks
  - Scientific computing: Cholesky decomposition
6. Example: Convolutional 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,2*y-2,2*x-2]*W2[0,0] + ...
                B[i,2*y-1,2*x-2]*W2[1,0] + ...
      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]

// 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]
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[1,0] + ... 
            A[i,y,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  /* Reduction */
    E[i] += T[i,j];
for j = 0 to Sj-1  /* Parallel loop */
    F[i,j] = T[i,j]/E[i]

Outer loop parallelism
Fusion of multi-dimensional permutable loops for improved locality
Example: Cholesky Decomposition

```c
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]);
}
```

2D parallelism:  \( p1 = i; p2 = j \)

```c
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]);
    }
}
```

Transformed Space: 3D pyramid

Many non-perfectly nested loops
3-deep fully permutable loop nest
with 2-dimensional parallelism
Blocking \( \rightarrow \) high performance
Summary: Two Key Algorithms

Loops

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

Processor ID

Array

\[ i_1 \leq i_2 \]

Loops

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

Time Stage
General Lessons

• Elegant mathematical approach
  – Exploit regularity in affine array accesses with affine mappings
  – Better performance, easier to get a correct compiler

• Coarse-grain parallelism: canonical representation of parallelism
  – Can block to tailor the code for specific machine architecture:
    • instruction-level parallelism, SIMD operations, cache/register locality

• Compiler advantage: Can customize for different machine models