Lecture 9
Basic Parallelization

I. Introduction
II. Data Dependence Analysis
III. Loop Nests
IV. Beyond Data Dependences
V. Blocking

Chapter 11.1-11.1.4

Machine Learning is Expensive

1 ExaFLOPS = 10^{18} FLOPS
Nvidia Volta GV100 GPU

80 Stream Multiprocessors
In each stream multiprocessor
- 64 FP32 cores
- 64 int cores
- 32 FP64 cores
- 8 Tensor cores

Tensor Cores
D = A \times B + C;
A, B, C, D are 4x4 matrices
4 \times 4 \times 4 matrix processing array
1024 floating point ops / clock

FP32: 15 TFLOPS
FP64: 7.5 TFLOPS
Tensor: 120 TFLOPS

Why Automatic Parallelization?

- Domains: scientific applications, signal processing, machine learning
  - Simpler but very useful domains
  - Has dense matrices
  - Lots of parallelism, ways to parallelize
  - But still hard to get good performance
  - Machine dependent

- Understanding parallelization makes you a better programmer for parallel machines

- Beautiful abstraction: linear algebra, integer linear programming
Parallelization of Numerical Applications

- **DoAll loop parallelism**
  - Find loops whose iterations are independent
  - Number of iterations typically scales with the problem
  - Usually much larger than the number of processors in a machine
  - Divide up iterations across machines

- **Doacross loop parallelism** and Loop Transformations in the next class

### Iteration Space

```plaintext
FOR i = 0 to 5
    FOR j = i to 7
        ...
```

- n-deep loop nests: n-dimensional polytope
- Iterations: coordinates in the iteration space
- Assume: iteration index is incremented in the loop
- Sequential execution order: lexicographic order
  - [0,0], [0,1], ..., [0,6], [0,7], [1,1], ..., [1,6], [1,7], ...

Basic Parallelism

Examples:

FOR i = 1 to 100
    \( A[i] = B[i] + C[i] \)

FOR i = 11 TO 20
    \( a[i] = a[i-1] + 3 \)

FOR i = 11 TO 20
    \( a[i] = a[i-10] + 3 \)

• Does there exist a data dependence edge between two different iterations?
• A data dependence edge is loop-carried if it crosses iteration boundaries
• DoAll loops: loops without loop-carried dependences

Affine Array Accesses

• Common patterns of data accesses: \((i, j, k \text{ are loop indexes})\)
  \( A[i,j], A[i-1, j+1]\)

• Domain of data dependence analysis
  – Array indexes are affine expressions of surrounding loop indexes
    • Loop indexes: \(i_0, i_1, \ldots, i_l\)
    • Integer constants: \(c_0, c_1, \ldots, c_0\)
    • Array index: \(c_0i_0 + c_1i_1 + \ldots + c_li_l + c_0\)
    • Affine expression: linear expression + a constant term \((c_0)\)
  – Loop bounds are affine expressions of outer loop indexes
  – Extend indexes to include symbolic constants: variables with constant values inside the loops
II. Recall: Data Dependences

- True dependence:
  \[ a = a \]

- Anti-dependence:
  \[ a = a \]

- Output dependence
  \[ a = a \]

Formulating Data Dependence Analysis

```plaintext
FOR i := 2 to 5 do
```

- Between read access `A[1]` and write access `A[i-2]` there is a dependence if:
  - there exist two iterations \( i_r \) and \( i_w \) within the loop bounds, s.t.
  - iterations \( i_r \) & \( i_w \) read & write the same array element, respectively
  \[ 3 \text{integers } i_w, i_r, \ 2 \leq i_w, i_r \leq 5 \ 
  i_r = i_w - 2 \]

- Between write access `A[i-2]` and write access `A[i-2]` there is a dependence if:
  \[ 3 \text{integers } i_w, i_r, \ 2 \leq i_w, i_r \leq 5 \ 
  i_w - 2 = i_r - 2 \]

- To rule out the case when the same instance depends on itself:
  - add constraint \( i_w \neq i_r \)
Memory Disambiguation

is

Undecidable at Compile Time

\[ \text{read}(n) \]

For \( i = \)

\[ a[i] = a[n] \]

Domain of Data Dependence Analysis

- Only use loop bounds and array indexes that are affine functions of loop variables and symbolic constants
  
  \[
  \text{for } i = 1 \text{ to } n \\
  \text{for } j = 2i \text{ to } 100 \\
  a[i+2j+3][4i+2j][i+1] = \ldots \\
  \ldots = a[1][2i+1][j]
  \]

- Assume a data dependence between the read & write operation if:
  - Let a read instance be denoted with indexes \( i_r, j_r \)
  - A write instance be denoted with indexes \( i_w, j_w \)

\[ \exists \text{Integers } i_r, j_r, i_w, j_w, n \]

\[
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 1 \\
-2 & 1 & 0 \\
0 & -1 & 0
\end{bmatrix} \cdot 
\begin{bmatrix}
\ldots \\
\ldots \\
\ldots \\
\ldots
\end{bmatrix} 
= 
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 1 \\
-2 & 1 & 0 \\
0 & -1 & 0
\end{bmatrix} \cdot 
\begin{bmatrix}
\ldots \\
\ldots \\
\ldots \\
\ldots
\end{bmatrix}
\]
Domain of Data Dependence Analysis

- Equate each dimension of array access; ignore non-affine ones
  - No solution \Rightarrow No data dependence
  - Solution \Rightarrow there may be a dependence

Complexity of Data Dependence Analysis

For every pair of accesses not necessarily distinct \((F_1, f_1)\) and \((F_2, f_2)\)

one must be a write operation

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

\[\exists\ \text{integers } i_1, i_2 \quad B_{i1} + b_1 \geq 0, B_{i2} + b_2 \geq 0\]

\[F_{i1} + f_1 = F_{i2} + f_2\]

If the accesses are not distinct, then add the constraint \(i_1 \neq i_2\)

- Equivalent to integer linear programming

\[\exists\ \text{integer } i \quad A_{1i} \leq b_1, \quad A_{2i} \leq b_2\]

- Integer linear programming is \textbf{NP-complete}
  - \(O(\text{size of the coefficients})\) or \(O(n^4)\)
Data Dependence Analysis Algorithm

- Typically solving many tiny, repeated problems
  - Integer linear programming packages optimize for large problems
  - Use memoization to remember the results of simple tests

- Apply a series of relatively simple tests
  - GCD: $2i, 2i+1$; GCD for simultaneous equations
  - Test if the ranges overlap

- Backed up by a more expensive algorithm
  - Use Fourier-Motzkin Elimination to test if there is a real solution
    - Keep eliminating variables to see if a solution remains
    - If there is no solution, then there is no integer solution

Fourier-Motzkin Elimination

- To eliminate a variable from a set of linear inequalities.
- To eliminate a variable $x_i$
  - Rewrite all expressions in terms of lower or upper bounds of $x_i$
  - Create a transitive constraint for each pair of lower and upper bounds.
- Example: Let $L_i$, $U_i$ be lower bounds and upper bounds resp
  - To eliminate $x_i$:

\[
\begin{align*}
L_1(x_2, \ldots, x_n) &\leq x_1 \leq U_1(x_2, \ldots, x_n) \\
L_2(x_2, \ldots, x_n) &\leq x_1 \leq U_2(x_2, \ldots, x_n)
\end{align*}
\]
**Example**

FOR $i = 1$ to 5
FOR $j = i+1$ to 5
$A[i,j] = f(A[i,i], A[i-1,j])$

write

<table>
<thead>
<tr>
<th>$1 \leq i$</th>
<th>$1 \leq i'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i \leq 5$</td>
<td>$i' \leq 5$</td>
</tr>
</tbody>
</table>

read

<table>
<thead>
<tr>
<th>$i + 1 \leq j$</th>
<th>$i' + 1 \leq j'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j \leq 5$</td>
<td>$j' \leq 5$</td>
</tr>
</tbody>
</table>

1: Data dep between $A[i,j], A[i',j']$

- $i = i'\ldots$
- $j = j'\ldots$
- $i'+1 \leq i'\ldots$

2: Data dep between $A[i,j]$ and $A[i'-1,j']$

- $i = i' - 1 \Rightarrow i+1 = i'$
- $j = j'$

Substituting

| $1 \leq i+1$, $i+1 \leq 5$ |
| $i + 2 \leq j$, $j \leq 5$ |

Combining

| $1 \leq i; i \leq 4$ |
| $i \leq j-2; j \leq 5$ |

Eliminating $i$:

| $1 \leq 4$, $1 \leq j-2; j \leq 5$ |

Eliminating $j$:

| $3 \leq j$, $j \leq 5$ |

Data Dependence Analysis Algorithm

- Typically solving many tiny, repeated problems
  - Integer linear programming packages optimize for large problems
  - Use memoization to remember the results of simple tests

- Apply a series of relatively simple tests
  - GCD: $2*i, 2*i+1$; GCD for simultaneous equations
  - Test if the ranges overlap

- Backed up by a more expensive algorithm
  - Use Fourier-Motzkin Elimination to test if there is a real solution
    - Keep eliminating variables to see if a solution remains
    - Add heuristics to encourage finding an integer solution.
  - Create 2 subproblems if a real, but not integer, solution is found.
    - For example, if $x = .5$ is a solution, create two problems
      by adding $x < 0$ and $x \geq 1$ respectively to original constraint.
III. Parallelism in Loop Nests

- Matrix Multiplication:
  ```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];
      }
    }
  }
  ```

Degrees of Parallelism

- Matrix Multiplication:
  ```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];
      }
    }
  }
  ```
IV. Beyond Data Dependences

Privatization:
- Scalar
  for $i = 1$ to $n$
  $t = (A[i] + B[i]) / 2$;
  $C[i] = t \cdot t$;
- Array
  for $i = 1$ to $n$
  for $j = 1$ to $n$
  $t[j] = (A[i,j] + B[i,j]) / 2$;
  for $j = 1$ to $n$
  $C[i,j] = t[j] \cdot t[j]$;

Reduction:
for $i = 1$ to $n$
  $sum = sum + A[i]$;

Interprocedural Parallelization

- Interprocedural symbolic analysis
  - Find interprocedural array indexes which are affine expressions of outer loop indices
- Interprocedural parallelization analysis
  - Data dependence based on summaries of array regions accessed
    - If the regions intersect, there is no parallelism
  - Find privatizable scalar variables and arrays
  - Find scalar and array reductions
An Example

V. Blocking

Data Accessed

1000 = 1000 x 1000 1000 1000 1002000

32 32 1000 32 1000 65024
Experimental Results

Speedup

With Blocking
Without Blocking

Processors

Code Transform

- Original
  ```
  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];
      }
    }
  }
  ```

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

• Basic parallelization
  – Doall loop: loops with no loop-carried data dependences
  – Data dependence for affine loop indexes = integer linear programming

• Outer loop is more coarse-grain
  – Less barrier overhead, less interprocessor communication
  – Interprocedural analysis is useful for coarse-grain parallelism
    • Ask users for help on unresolved dependences

• Blocking is useful for uniprocessors and multiprocessors
  – Minimize interprocessor communication
  – Minimize cache misses
  – Applicable to entire memory hierarchy: virtual memory, registers
    • Tiny blocks to reuse registers and minimize memory fetches