Lecture 9
Basic Parallelization

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

Chapter 11.1-11.1.4

Machine Learning is Expensive

7 ExaFLOPS 60 Million Parameters
20 ExaFLOPS 300 Million Parameters
100 ExaFLOPS 8700 Million Parameters

2015 Microsoft ResNet Superhuman Image Recognition
2016 Baidu Deep Speech 2 Superhuman Voice Recognition
2017 Google Neural Machine Translation Near Human Language Translation

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 x 4 x 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

```
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 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_{n-1}, ..., i_1
    • Integer constants: c_0, c_{n-1}, ..., c_0
    • Array index: c_0i_0 + c_{n-1}i_{n-1} + ... + c_1i_1 + 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

FOR \( i := 2 \) to \( 5 \) do

\[ A[i-2] = A[i]+1; \]

- 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

\[ \exists \text{integers } i_w, i_r \quad 2 \leq i_w, i_r \leq 5 \quad i_r = i_w - 2 \]

- Between write access \( A[i-2] \) and write access \( A[i-2] \) there is a dependence if:

\[ \exists \text{integers } i_w, i_v \quad 2 \leq i_w, i_v \leq 5 \quad i_w - 2 = i_v - 2 \]

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

is

Undecidable at Compile Time

\[
\text{read}(n) \\
\text{For } i = \\
a[i] = a[n]
\]

Domain of Data Dependence Analysis

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

- 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 \( i_w, j_w \)

\[
\exists \text{Integers } i_r, j_r, i_w, j_w, n \\
\begin{bmatrix}
  1 & 0 & 0 \\
  -1 & 0 & 1 \\
  0 & -1 & 0 \\
  100 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  i_r \\
  j_r
\end{bmatrix}
= 
\begin{bmatrix}
  -1 & 0 & 0 \\
  0 & -1 & 0 \\
  0 & 0 & -100
\end{bmatrix}
\begin{bmatrix}
  i_w \\
  j_w
\end{bmatrix}
\begin{bmatrix}
  1 & 0 & 0 \\
  -1 & 0 & 1 \\
  0 & -1 & 0 \\
  0 & 0 & 100
\end{bmatrix}
\begin{bmatrix}
  i_r \\
  j_r
\end{bmatrix}
\begin{bmatrix}
  -1 & 0 & 0 \\
  0 & -1 & 0 \\
  0 & 0 & -100
\end{bmatrix}
\begin{bmatrix}
  i_w \\
  j_w
\end{bmatrix}
\begin{bmatrix}
  1 & 0 & 0 \\
  -1 & 0 & 1 \\
  0 & -1 & 0 \\
  0 & 0 & 100
\end{bmatrix}
\begin{bmatrix}
  i_r \\
  j_r
\end{bmatrix}
\]
Domain of Data Dependence Analysis

- Equate each dimension of array access; ignore non-affine ones
  - No solution → No data dependence
  - Solution → 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 \(B_1i_1 + b_1 \geq 0, B_2i_2 + b_2 \geq 0\) be the corresponding loop bound constraints,
\[ \exists \text{ integers } i_1, i_2 \quad B_1i_1 + b_1 \geq 0, B_2i_2 + b_2 \geq 0 \]
\[ F_1i_1 + f_1 = F_2i_2 + f_2 \]
If the accesses are not distinct, then add the constraint \(i_1 \neq i_2\)
- Equivalent to integer linear programming
  \[ \exists \text{ integer } i \quad A_1i \leq b_1, A_2i \geq b_2 \]
- Integer linear programming is \(\text{NP-complete}\)
  - \(O(\text{size of the coefficients})\) or \(O(n^c)\)
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 $x_i$:

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

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

```
Example

```

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
    - 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 $x \leq 0$ 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 * 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] * t[j];
  ```

Reduction:

```plaintext
for i = 1 to n
  sum = sum + A[i];
```
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