Lecture 9

Basic Parallelization

I. Introduction

II. Data Dependence Analysis Problem

III. Data Dependence Algorithm

Chapter 11.1-11.1.4
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

https://wccftech.com/nvidia-volta-tesla-v100-cards-detailed-150w-single-slot-300w-dual-slot-gv100-powered-pcie-accelerators/
Why Automatic Parallelization?

- Domains: signal processing, scientific applications, 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, automation provides portability

- Understanding automatic parallelization makes you a better programmer for parallel machines

- Beautiful abstraction: integer linear programming, linear algebra
Lecture Outline

• Goal: Direct parallelization (without loop transformations)
  – Can we execute all iterations in a loop nest in parallel?

• Outline
  – Find parallelism with data dependence analysis
    • Problem formulation
    • Algorithm
  – Find more parallelism with other analyses
    • Reduction
    • Privatization
    • Interprocedural analysis
Parallelization of Numerical Applications

- This class: Find DoAll loop parallelism in existing code
  - 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
Data Dependence Algorithm History

• Original approach
  – Many simple tests derived to handle patterns in programs
    • To prove independence but not dependence
  – But new patterns keep emerging → new papers keep emerging

• Dror E. Maydan, John L. Hennessy, Monica S. Lam: Efficient and Exact Data Dependence Analysis. PLDI 1991: 1-14
  – 1st to formulate data dependence as integer linear programming
  – Is this an overkill?
    • A very expensive algorithm; most programs have simple indices
  – Key idea:
    • Achieve generality for a well-defined domain (affine indices)
    • Optimize evaluation for the common patterns
    • Rely on general, but more expensive solution when necessary!

• Outcome: Data dependence analysis solved (no more papers)

  Lesson: Leverage general Maths abstraction
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 by 1 in the loop
- Sequential execution order: lexicographic order
  - [0,0], [0,1], ..., [0,6], [0,7],
  - [1,1], ..., [1,6], [1,7],
  - ...

CS243: Parallelization
7
M. Lam
Basic Parallelism

Examples:

FOR i = 1 to 100

FOR i = 11 TO 20

FOR i = 11 TO 20

• 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 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: \(i_n, i_{n-1}, \ldots, i_1\)
    • Integer constants: \(c_n, c_{n-1}, \ldots, c_0\)
    • Array index: \(c_n i_n + c_{n-1} i_{n-1} + \ldots + c_1 i_1 + c_0\)
    • 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
Recall: Data Dependences

- **True dependence:**
  
  \[
  a = \\
  = a
  \]

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

- **Output dependence**
  
  \[
  a = \\
  a =
  \]
Formulating Data Dependence Analysis

\textbf{FOR} \( i := 2 \) \textbf{to} 5 \textbf{do}
\[ A[i-2] = A[i]+1; \]

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

- Between read access \( A[i] \) 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

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

```
for i = 1 to n
    for j = 2i to 100
        a[i+2j+3][4i+2j][i*i] = ...
        ... = a[1][2i+1][j]
```

• Equate each dimension of array access; ignore non-affine ones
  – No solution \(\Rightarrow\) No data dependence
  – Solution \(\Rightarrow\) there may be a dependence
Domain of Data Dependence Analysis

- Only use loop bounds and array indexes that are affine functions of loop variables and symbolic constants

\[
\text{for } i = 1 \text{ to } n \\
\text{for } j = 2i \text{ to } 100 \\
a[i+2j+3][4i+2j][i*i] = ... \\
... = a[1][2i+1][j]
\]

Let a read instance be denoted with indexes \(i_r, j_r\) and a write instance be denoted with indexes \(i_w, j_w\).

Assume a data dependence between the read & write operation if:

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

\[
\begin{align*}
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 1 \\
-2 & 1 & 0 \\
0 & -1 & 0
\end{bmatrix}
& \begin{bmatrix}
i_r \\
-1 \\
0 \\
100
\end{bmatrix}
& \begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
& \begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 1 \\
-2 & 1 & 0 \\
0 & -1 & 0
\end{bmatrix}
& \begin{bmatrix}
i_r \\
-1 \\
0 \\
100
\end{bmatrix}
& \begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
\end{align*}
\]

loop bounds on write

loop bounds on read

same memory location

CS243: Parallelization
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,

\[
\exists \text{ integers } i_1, i_2 \quad B_1i_1 + b_1 \geq 0, \quad B_2i_2 + b_2 \geq 0
\]

\[
F_1i_1 + f_1 = F_2i_2 + f_2
\]

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

• Equivalent to integer linear programming

\[
\exists \text{ integer } i \quad A_1i \leq b_1 \quad A_2i = b_2
\]

• Integer linear programming is NP-complete
  – \(O(\text{size of the coefficients})\) or \(O(n^n)\)
Lecture Outline

- Goal: Simple parallelization
  - Can we execute all iterations in a loop nest in parallel?

- Outline
  - Find parallelism with data dependence analysis
    - Problem formulation
    - Algorithm
Data Dependence Analysis Algorithm

• Typically solving many tiny, repeated problems
  – Integer linear programming packages optimize for large problems
    • Inappropriate for data dependence analysis
  – Use memoization to remember the results of tests performed

• Apply a series of relatively simple tests (11.6.2)
  – Example 1: GCD: 2*i, 2*i+1; GCD for simultaneous equations
  – Example 2: 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 real solution, then there is no integer solution
Fourier-Motzkin Elimination

- To eliminate a variable from a set of linear inequalities.
- To eliminate a variable $x_1$
  - Rewrite all expressions in terms of lower or upper bounds of $x_1$
  - Create a transitive constraint for each pair of lower and upper bounds.
- Example: Let $L, U$ be lower bounds and upper bounds resp
  - To eliminate $x_1$: 

\[
L_1(x_2, \ldots, x_n) \leq x_1 \leq U_1(x_2, \ldots, x_n) \\
L_2(x_2, \ldots, x_n) \leq x_1 \leq U_2(x_2, \ldots, x_n)
\]

\[
L_1(x_2, \ldots, x_n) \leq U_1(x_2, \ldots, x_n) \\
L_1(x_2, \ldots, x_n) \leq U_2(x_2, \ldots, x_n) \\
L_2(x_2, \ldots, x_n) \leq U_1(x_2, \ldots, x_n) \\
L_2(x_2, \ldots, x_n) \leq U_2(x_2, \ldots, x_n)
\]
Example

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

write
$1 \leq i$
$i \leq 5$
$i + 1 \leq j$
$j \leq 5$

read
$1 \leq i'$
$i' \leq 5$
$i' + 1 \leq j'$
$j' \leq 5$

1. Writes: $A[i,j]$ trivially has no dependence
2. Data dep between $A[i,j], A[i',i']$
   $i = i'$
   $j = i'$
   $i'+1 \leq i'$
   $1 \leq 0$
   
   Quiz: is there a dependence?

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

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

Substituting $i', j'$ in read loop bounds
$1 \leq i + 1, i + 1 \leq 5$
$i + 2 \leq j, j \leq 5$

Combining loop bounds for write & read
$1 \leq i; i \leq 4, i \leq j - 2; j \leq 5$

Eliminating $i$ using Fourier-Motzkin:
$1 \leq 4; 1 \leq j - 2; j \leq 5$
$3 \leq j; j \leq 5$

Eliminating $j$ using Fourier-Motzkin:
$3 \leq 5$

Quiz: is there a dependence?
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
    • Use heuristics to find an integer solution.
  – Create 2 subproblems if a real, but not integer, solution is found.
    • For example, k = .5 is a solution, (no integer solutions)
      create two problems, by adding k ≤ 0 and k ≥ 1 respectively to original constraint.

In practice, we rarely need Fourier-Motzkin Elimination, and never have to split the problem
Lecture Outline

• Goal: Simple parallelization
  – Can we execute all iterations in a loop nest in parallel?

• Outline
  – Find parallelism with data dependence analysis
    • Problem formulation
    • Algorithm
  – Find more parallelism with other analyses
    • Reduction
    • Privatization
    • Interprocedural analysis
Privatization and Reduction

Privatization:

for i = 1 to n  // scalar
    t = (A[i] + B[i]) / 2;
    C[i] = t * t;

for i = 1 to n  // array
    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:

for i = 1 to n  // scalar
    sum = sum + A[i];

for i = 1 to n  // array
    for j = 1 to n
        sum[i] = sum[i] + B[i,j];
Performance of Parallel Programs

• What determines the performance of parallelized programs
  – Amdahl’s Law
    • Suppose 90% of a program is parallelized, what is the limit of the parallelism speed up?
  – Overhead of parallelism:
    • Synchronization overhead:
      – DoAll loops have “barriers”; processors wait for all iterations to finish.
    • Communication overhead:
      – Movement of dependent data across processors

• Granularity of parallelism
  – Fine-grain: parallelism in innermost loops
  – Coarse-grain: parallelism in outermost loops

Quiz: Which is better?
Example: 2D FFT

> 1000 lines of code
Multiple levels of calls

Parallelizing innermost loops will slow down the program!
Outermost loop parallelized (shown in red)
Requires interprocedural parallelization!

Call Graph Representation
Node: Function
Edge: Caller to callee
Empirical Research Process

• University of Illinois [Kuck et al.] 1989
  – Perfect Club benchmark: hand-parallelized programs
  – 13 Fortran programs, 50,000 lines of code,

• Stanford SUIF parallelizer (Lam et al.) 1995
  – Implemented known techniques to discover missing techniques
    • There are many affine array indices
    • Found a lot of inner loops – ran slower if parallelized
    • Identify what was needed
  – Developed interprocedural array reduction/privatization
  – Breakthrough:
    Parallelized a huge loop not previously known to be parallel
  – A single missed dependence renders loop not parallelizable
    • E.g. a random number generator
    • Ask users for help on unresolved dependences
Conclusions

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

• General Lesson
  – Formulation of data dependence analysis as ILP
    • A general math framework
    • Put an end to years of incremental patches to heuristics
to handle cases discovered in practice