Lecture 9
Basic Parallelization

I. Introduction
II. Data Dependence Analysis
III. Loop Nests + Locality
IV. Interprocedural Parallelization

Chapter 11.1-11.1.4

Machine Learning is Expensive

1 ExaFLOPS = 10^{18} FLOPS
A Stream Processor

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

\[ \text{FOR } i = 0 \text{ to } 5 \]
\[ \text{FOR } j = i \text{ to } 7 \]

- 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
  - \([0,0], [0,1], \ldots, [0,6], [0,7], [1,1], \ldots, [1,6], [1,7], \ldots\)
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_n, i_n-1, ..., i_1
    • Integer constants: c_n, c_n-1, ..., c_0
    • Array index: c_n i_n + c_n-1 i_n-1 + ... + 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
II. Recall: Data Dependences

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

- **Anti-dependence:**

  \[
  a = a = a
  \]

- **Output dependence**

  \[
  a = a = a
  \]

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

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

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

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

  - 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

- 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 $B_1i_1+b_1 \geq 0, B_2i_2+b_2 \geq 0$ be the corresponding loop bound constraints,

$\exists$ integers $i_1, i_2$

$B_1i_1 + b_1 \geq 0, B_2i_2 + b_2 \geq 0$

$F_1i_1 + f_1 = F_2i_2 + f_2$

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

- Equivalent to integer linear programming

$\exists$ integer $i$

$A_1i \preceq b_1, A_2i \preceq b_2$

- Integer linear programming is NP-complete

- $O$(size of the coefficients) or $O(n^3)$
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
    - 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 \( x_1 \)
  - 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 \( x_1 \):

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

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

III. Parallelism in Loop Nests

• Matrix Multiplication:
  
  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];
      }
    }
  }
  ```

Blocking

| Data Accessed | 1002000 | 65024 |
### Experimental Results

- **With Blocking**
- **Without Blocking**

![Graph showing speedup vs. processors](image)

### Code Transform

- **Before**
  ```java
  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];
      }
    }
  }
  ```

- **After**
  ```java
  for (ii = 0; ii < n; ii++) {
    for (jj = 0; jj < n; jj++) {
      for (kk = 0; kk < n; kk++) {
        for (i = ii; i < min(n,kk+B); i++) {
          for (j = jj; j < min(n,kk+B); j++) {
            for (k = kk; k < min(n,kk+B); k++) {
              Z[i,j] = Z[i,j] + X[i,k] * Y[k,j];
            }
          }
        }
      }
    }
  }
  ```
SIMD: Innermost Parallelism

• **Before**

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

• **After**

```c
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++) {
            for (j' = j < min(n,j'+b); j'++) {
              Z[i,j'] = Z[i,j'] + X[i,k] * Y[k,j'];
            }
          }
        }
      }
    }
  }
}
```

**Observations**

• **Outer loop is more coarse-grain**
  - Less barrier overhead, less interprocessor communication

• **Any parallel loop can be blocked and moved to an inner loop**
  - Transformations will be explained in the next class

• **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
**Canonical Form**

- **Nests of outermost maximum degrees of parallelism**
  - Outer loop parallelism can always be made inner loop parallelism
  - Machine independent
  - Transformations can increase outermost maximum degrees of parallelism (next class)

**A High-Level Strategy**

- **Multiprocessors:** Find the outermost maximum degrees of parallelism
  - Assign multi-dimensional blocks to processors
  - To minimize barrier and processor communication overhead
- Further block for caches and registers
- For SIMD operations
  - Block/Stripmine a parallel loop to create an innermost loop that reads data contiguously
IV. Whole Numerical Applications

• Amdahl’s Law

• Kernels
  – A small number of tight loops with large data set sizes
  – Extremely important to optimize for locality

• Across kernels:
  – Important when
    • Kernel parallelism does not dominate the computation
    • Or kernel parallelism cannot occupy the whole machine
  – Two techniques
    • Interprocedural analysis of sequential programs
    • Parallel execution of a high-level data flow graph (TensorFlow)

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 do not intersect, there is no parallelism
  – Find privatizable scalar variables and arrays
  – Find scalar and array reductions
Conclusions

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

- Locality is important for unprocessors and multiprocessors

- Canonical form: Nests of outermost coarse-grain parallelism
  - Transform the code according to machine characteristics

- Interprocedural analysis is useful for affine indices
  - Ask users for help on unresolved dependences