COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515
Course Information

- **Meeting time:** TTh 09:25am - 10:40am
- **Meeting place:** Keck 107
- **Instructor:** Prof. Vivek Sarkar
- **Assistant Instructor:** Dr. Jun Shirako
- **Web site:** [https://wiki.rice.edu/confluence/display/PARPROG/COMP515](https://wiki.rice.edu/confluence/display/PARPROG/COMP515)
- **Prerequisite:** COMP 412
- **Textbook**
- **Grading rubric**
  - Homeworks (30%)
  - Exam 1 (20%), Exam 2 (20%)
  - Class project (30%)
- **Acknowledgment:** Slides from previous offerings of COMP 515 by Prof. Ken Kennedy ([http://www.cs.rice.edu/~ken/comp515/](http://www.cs.rice.edu/~ken/comp515/))
Dependence-Based Compilation

• Vectorization and Parallelization require a deeper analysis than optimization for scalar machines
  — Must be able to determine whether two accesses to the same array might be to the same location

• Dependence is the theory that makes this possible
  — There is a dependence between two statements if they might access the same location, there is a path from one to the other, and one access is a write

• Dependence has other applications
  — Memory hierarchy management—restructuring programs to make better use of cache and registers
    - Includes input dependences
  — Scheduling of instructions
Syllabus

• Introduction

• Dependence Theory and Practice

• Preliminary Transformations
  — Loop normalization, scalar data flow analysis, induction variable substitution, scalar renaming.
Syllabus (contd)

• **Fine-Grain Parallel Code Generation**
    The layered vector code-generation algorithm and its complexity.
    Loop interchange.

• **Unimodular & Polyhedral loop transformation frameworks**
  – New topics not covered in textbook

• **Coarse-Grain Parallel Code Generation**
  – Loop Interchange. Loop Skewing. Scalar and array expansion.
    dependence tests. Parallel code generation and its problems.

• **Control Dependence**
  – Types of branches. If conversion. Control dependence. Program
    dependence graph.
Syllabus (contd)

• **Memory Hierarchy Management**
  - The use of dependence in scalar register allocation and management of the cache memory hierarchy.

• **Scheduling for Superscalar and Parallel Machines**
  - Role of dependence. List Scheduling. Software Pipelining. Work scheduling for parallel systems. Guided Self-Scheduling

• **Interprocedural Analysis and Optimization**

• **Compilation of Other Languages.**
  - C, Verilog, Fortran 90, HPF.
Compiler Challenges for High Performance Architectures

Allen and Kennedy, Chapter 1
Features of Machine Architectures

• Pipelining
• Multiple execution units
  — pipelined
• Vector operations
  — includes fine-grained SIMD vector parallelism
• Parallel processing
  — Multicore, shared memory, distributed memory, message-passing
• Superscalar instruction issue, software/hardware prefetch
• Registers
• Memory hierarchy
• Combinations of the above
Instruction Pipelining

- Instruction pipelining
  - DLX Instruction Pipeline

- What is the performance challenge?
Replicated Execution Logic (Floating Point Adders)

• Pipelined Execution Units

• Multiple Execution Units

What is the performance challenge?
Vector Operations

- Apply same operation to different positions of one or more arrays
  - Goal: keep pipelines of execution units full
  - Example:
    
    
    \[
    \begin{align*}
    &\text{VLOAD} & V1, A \\
    &\text{VLOAD} & V2, B \\
    &\text{VADD} & V3, V1, V2 \\
    &\text{VSTORE} & V3, C
    \end{align*}
    \]
Very Large Instruction Word (VLIW)

- Multiple instruction issue on the same cycle
  - Wide word instruction (or superscalar)
  - Designated functional units for instruction slots

Source: “VLIW/EPIC: Statically Scheduled ILP”, Joel Emer
SIMD (Single Instruction Multiple Data)

- Short SIMD architectures
  - E.g., MMX, SSE, AltiVec
  - Limited vector length (16 bytes for AltiVec)
  - Contiguous memory access
  - Data alignment constraint (128-bit alignment for AltiVec)
SIMT (Single Instruction Multiple Thread)

- SIMT: Single-Instruction Multi-Thread executes one instruction across many independent threads
  - **Warp**: a set of 32 parallel threads that execute a SIMT instruction
  - SIMT provides easy single-thread scalar programming with SIMD efficiency

- Hardware implements zero-overhead warp and thread scheduling

- SIMT threads can execute independently
  - SIMT warp diverges and converges when threads branch independently
  - Best efficiency and performance when threads of a warp execute together

SMP Parallelism (Homogeneous Multicore)

- Multiple processors with uniform shared memory
  - Task Parallelism
    - Independent tasks
  - Data Parallelism
    - the same task on different data

• What is the performance challenge?
Distributed Memory

- **Memory packaged with processors**
  - Message passing
  - Distributed shared memory

- **SMP clusters**
  - Shared memory on node, message passing off node

- **Distributed memory in multicore processors**
  - Intel Single Chip Cloud (SCC) computer
  - Tilera

- **What are the performance issues?**
  - Minimizing communication
    - Data placement
  - Optimizing communication
    - Aggregation
    - Overlap of communication and computation
Compiler Technologies

• Program Transformations
  — Many of these architectural issues can be dealt with by restructuring transformations that can be reflected in source
    - Vectorization, parallelization, cache reuse enhancement
  — Two key challenges:
    - Determining when transformations are legal
    - Selecting transformations based on profitability

• Low level code generation
  — Some issues must be dealt with at a low level
    - Prefetch insertion
    - Instruction scheduling

• All require some understanding of the ways that instructions and statements depend on one another (share data)
Fortran DO loop notation

\begin{verbatim}
DO I = 1, N
  . . .
END DO
\end{verbatim}

and

\begin{verbatim}
DO 10 I = 1, N
  . . .
END DO
10 CONTINUE
\end{verbatim}

are equivalent to the following C-for loop:

\begin{verbatim}
for(I = 1; I <= N; I++) {
  . . .
}
\end{verbatim}
# Fortran column major vs. C row-major data layouts

<table>
<thead>
<tr>
<th>Fortran: real*8</th>
<th>C: double</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(100,100)</td>
<td>A[100][100]</td>
</tr>
<tr>
<td>A(1,1)</td>
<td>A[0][0]</td>
</tr>
<tr>
<td>A(2,1)</td>
<td>A[0][1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A(100,1)</td>
<td>A[0][99]</td>
</tr>
<tr>
<td>A(1,2)</td>
<td>A[1][0]</td>
</tr>
<tr>
<td>A(2,2)</td>
<td>A[1][1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A(100,2)</td>
<td>A[1][99]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
A Common Problem: Matrix Multiply

\[
\begin{align*}
&\text{DO } I = 1, N \\
&\quad \text{DO } J = 1, N \\
&\quad \quad C(J,I) = 0.0 \\
&\quad \text{DO } K = 1, N \\
&\quad \quad C(J,I) = C(J,I) + A(J,K) \times B(K,I) \\
&\quad \text{ENDDO} \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]
DO I = 1, N,
   DO J = 1, N, 4 // Unroll J loop 4 times
      C(J,I) = 0.0    !Register 1
      C(J+1,I) = 0.0  !Register 2
      C(J+2,I) = 0.0  !Register 3
      C(J+3,I) = 0.0  !Register 4
   DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
      C(J+1,I) = C(J+1,I) + A(J+1,K) * B(K,I)
      C(J+2,I) = C(J+2,I) + A(J+2,K) * B(K,I)
      C(J+3,I) = C(J+3,I) + A(J+3,K) * B(K,I)
   ENDDO
ENDDO
ENDDO
Problems for Vectors

- Inner loop must be vector
  - And should be stride 1
  - Note that array layout is column-major for FORTRAN and row-major in C

- Vector registers have finite length (Cray: 64 elements, modern SIMD processor operands are in the 256-512 byte range)
  - Would like to reuse vector register in the compute loop

- Solution
  - Strip mine the loop over the stride-one dimension to 64
  - Move the iterate over strip loop to the innermost position
    - Vectorize it there
Vectorizing Matrix Multiply

DO I = 1, N
    DO J = 1, N, 64
        DO JJ = J, J+63
            C(JJ,I) = 0.0
        ENDDO
    ENDDO
ENDDO

DO K = 1, N
    C(JJ,I) = C(JJ,I) + A(JJ,K) * B(K,I)
ENDDO
Vectorizing Matrix Multiply

DO I = 1, N
    DO J = 1, N, 64
        DO JJ = J, J+63
            C(JJ,I) = 0.0
        ENDDO
    DO K = 1, N
        DO JJ = J, J+63
            C(JJ,I) = C(JJ,I) + A(JJ,K) * B(K,I)
        ENDDO
    ENDDO
ENDDO
MatMult for a Vector Machine
(using array language notation)

DO I = 1, N
  DO J = 1, N, 64
    C(J:J+63,I) = 0.0
    DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K)*B(K,I)
    ENDDO
  ENDDO
ENDDO
Matrix Multiply on Parallel SMPs

\[
\begin{align*}
\text{DO } I & = 1, N \quad ! \text{ Independent for all } I \\
\text{DO } J & = 1, N \\
\quad C(J,I) & = 0.0 \\
\text{DO } K & = 1, N \\
\quad C(J,I) & = C(J,I) + A(J,K) \times B(K,I) \\
\end{align*}
\]

ENDDO
ENDDO
ENDDO
Bernstein’s Conditions [1966]

• When is it safe to run two tasks R1 and R2 in parallel?
  – If none of the following holds:
    1. R1 writes into a memory location that R2 reads
    2. R2 writes into a memory location that R1 reads
    3. Both R1 and R2 write to the same memory location

• How can we apply this to loop parallelism?
  • Think of loop iterations as tasks

• How can we apply this to statement-level parallelism?
  • Think of statement instances as tasks

• Time for Worksheet #1!
Problems on a Parallel Machine

• Parallelism must be found at the outer loop level
  — But how do we know?

• Solution
  — Bernstein's conditions
    - Can we apply them to loop iterations?
    - Yes, with dependence
  — Statement S2 depends on statement S1 if
    - S2 comes after S1
    - S2 must come after S1 in any correct reordering of statements
  — Usually keyed to memory
    - Path from S1 to S2
    - S1 writes and S2 reads the same location
    - S1 reads and S2 writes the same location
    - S1 and S2 both write the same location
MATMUL on a Shared-Memory MP

\[
\begin{align*}
\text{PARALLEL DO } & I = 1, N \\
& \text{DO } J = 1, N \\
& \quad C(J,I) = 0.0 \\
& \quad \text{DO } K = 1, N \\
& \quad \quad C(J,I) = C(J,I) + A(J,K) \times B(K,I) \\
& \quad \text{ENDDO} \\
& \text{ENDDO} \\
& \text{END PARALLEL DO}
\end{align*}
\]
MatMult on a Vector SMP

PARALLEL DO I = 1, N
   DO J = 1, N, 64
      C(J:J+63,I) = 0.0
   DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K)*B(K,I)
   ENDDO
   ENDDO
ENDDO
Memory Hierarchy

• Problem: memory is moving farther away in processor cycles
  — Latency and bandwidth difficulties

• Solution
  — Reuse data in cache and registers

• Challenge: How can we enhance reuse?
  — Fortran example
    
    DO I = 1, N
    DO J = 1, N
      C(I) = C(I) + A(J)
    
    — Equivalent C/Java code
    
    for (int I = 1; I <= N; I++)
      for (int J = 1; J <= N; J++)
        C[I] = C[I] + A[J];
  — Strip mining to reuse data from cache
Example of Cache Reuse

\[
\begin{align*}
\text{DO } & I = 1, N \\
\text{DO } & J = 1, M \\
& C(I) = A(I) + B(J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

• J loop reuses \( C(I) \) and \( A(I) \), but not \( B(J) \)
• I loop reuses \( B(J) \), but not \( C(I) \) and \( A(I) \)
• Solution
  — Block/tile the loops so you get reuse of both \( A \) and \( B \)
    - Multiply a block of \( A \) by a block of \( B \) and add to block of \( C \)
  — When is it legal to interchange the iterate over block loops to the inside?

• Time for Worksheet #2!
Course Project Logistics

• Goal of course project is to perform an in-depth study of a research problem related to the course
  — Should include a theoretical focus with a project report (with references to recent related work)
  — Practicality can be demonstrated using hand-coded transformations, or using a tool such as ISL or LoopPy to perform the transformations

• Project should be done in groups of 2 or 3
  — Project groups should be finalized by the end of this week
  — One-on-one meetings will be scheduled with groups and instructors next week to discuss project topics

• Final project presentations scheduled in class on Dec 1st and Dec 3rd
Worksheet #1

Name: _____________________    Netid: _____________________

DO  I = 1, N
    T = A[I]                S1
    B[I] = T                S3
ENDDO

• Using Bernstein conditions, identify pairs of statement instances that can exhibit one of the following conditions (a different pair for each condition)
  1. R1 writes into a memory location that R2 reads
  2. R2 writes into a memory location that R1 reads
  3. Both R1 and R2 write to the same memory location
  4. None of the above
Worksheet #2

Name: _____________________    Netid: ____________________

DO I = 1, N
   DO J = 1, M
      C(I) = A(I) + B(J)
   ENDDO
ENDDO

1. Assuming a uniprocessor cache with one word per cache line, and unbounded ("infinite") capacity, how many cache misses (memory accesses) are incurred by the above code?

2. How does your answer change if the cache can only hold 4 words?