COMP 515: Advanced Compilation for Vector and Parallel Processors

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

• Meeting time: TTh 1:00pm - 2:20pm
• Meeting place: DH 3131
• Instructor: Vivek Sarkar
  — vsarkar@rice.edu, x5304, DH 3131
• Web site: http://www.cs.rice.edu/~vsarkar/comp515
• Prerequisites
  — Required: COMP 314, COMP 221
  — Recommended: COMP 412
• Text
• Course Requirements:
  — Homeworks (25%).
  — Midterm Examination (25%).
  — Final Examination (25%).
  — Class Presentations (25%).
Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - 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**

• **Coarse-Grain Parallel Code Generation**

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

• **Scheduling for Superscalar and Parallel Machines**  

• **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
- Parallel processing
  - Shared memory, distributed memory, message-passing
- VLIW and Superscalar instruction issue
- Registers
- Cache hierarchy
- Combinations of the above
  - Parallel-vector machines
Instruction Pipelining

- Instruction pipelining
  - DLX Instruction Pipeline

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

- **Pipelined Execution Units**

  ![Pipelined Execution Units Diagram](image)

  - Inputs: Fetch Operands (FO), Equate Exponents (EE), Add Mantissas (AM), Normalize Result (NR)

- **Multiple Execution Units**

  ![Multiple Execution Units Diagram](image)

  - Adders: Adder 1 (b1 + c1), Adder 2 (b2 + c2), Adder 3 (b3 + c3), Adder 4 (b4 + c4)

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:
      
      \[
      \begin{align*}
      VLOAD & \quad V1,A \\
      VLOAD & \quad V2,B \\
      VADD & \quad V3,V1,V2 \\
      VSTORE & \quad V3,C
      \end{align*}
      \]
Very Large Instruction Word (VLIW)

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

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no x-operation RAW check
  - No data use before data ready => no data interlocks

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?
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 convert this to loop parallelism?
  - Think of loop iterations as tasks

- Does this apply to sequential loops embedded in an explicitly parallel program?
  - Impact of memory model on ordering of read operations
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?
  - *Coloring register allocation works well*
    - But only for scalars
      DO I = 1, N
      DO J = 1, N
      C(I) = C(I) + A(J)
  - Strip mining to reuse data from cache
Distributed Memory

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

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

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

• Program Transformations
  — Most of these architectural issues can be dealt with by restructuring transformations that can be reflected in source
    - Vectorization, parallelization, cache reuse enhancement
  — 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)
A Common Problem: Matrix Multiply

\begin{verbatim}
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
\end{verbatim}
MatMult for a Pipelined Machine

DO  I = 1, N,
    DO  J = 1, N, 4
        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

• Vector registers have finite length (Cray: 64 elements)
  — 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 = 0, 63
         C(JJ, I) = 0.0
      ENDDO
      DO K = 1, N
         C(J, I) = C(J, I) + A(J, K) * B(K, I)
      ENDDO
   ENDDO
ENDDO
Vectorizing Matrix Multiply

DO I = 1, N
  DO J = 1, N, 64
    DO JJ = 0, 63
      C(JJ,I) = 0.0
    ENDDO
  ENDDO
  DO K = 1, N
    DO JJ = 0, 63
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
MatMult for a Vector Machine

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

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

DO I = 1, N  ! Independent for all I
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
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
MatMult on a Shared-Memory MP

PARALLEL DO I = 1, N
    DO J = 1, N
        C(J,I) = 0.0
        DO K = 1, N
            C(J,I) = C(J,I) + A(J,K) * B(K,I)
        ENDDO
    ENDDO
ENDDO
END PARALLEL DO
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
Matrix Multiply for Cache Reuse

\[
\begin{align*}
\text{DO} & \quad I = 1, N \\
\text{DO} & \quad J = 1, N \\
& \quad C(J,I) = 0.0 \\
\text{DO} & \quad K = 1, N \\
& \quad C(J,I) = C(J,I) + A(J,K) \times B(K,I) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Problems on Cache

- There is reuse of C but no reuse of A and B
- Solution
  - Block 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?
MatMult on a Uniprocessor with Cache

DO I = 1, N, S
  DO J = 1, N, S
    DO p = I, I+S-1
      DO q = J, J+S-1
        C(q,p) = 0.0
      ENDDO
    ENDDO
  ENDDO
  DO K = 1, N, T
    DO p = I, I+S-1
      DO q = J, J+S-1
        DO r = K, K+T-1
          C(q,p) = C(q,p) + A(q,r) * B(r,p)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
ENDDO
ENDDO
ENDDO
ENDDO
ENDDO
Dependence

- **Goal:** aggressive transformations to improve performance
- **Problem:** when is a transformation legal?
  - Simple answer: when it does not change the meaning of the program
  - But what defines the meaning?
- **Same sequence of memory states**
  - Too strong!
- **Same answers**
  - Hard to compute (in fact intractable)
  - Need a sufficient condition
- **We use in this book:** dependence
  - Ensures instructions that access the same location (with at least one a store) must not be reordered
Summary

- Modern computer architectures present many performance challenges
- Most of the problems can be overcome by transforming loop nests
  - Transformations are not obviously correct
- Dependence tells us when this is feasible
  - Most of the book is about how to use dependence to do this
- Reading list for next lecture
  - Chapter 2, Dependence: Theory and Practice