Compiler Challenges for High Performance Architectures

Moore’s Law

• The observation made in 1965 by Gordon Moore, co-founder of Intel, said that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented.

• Moore predicted that this trend would continue for the foreseeable future.
  — In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore’s Law, which Moore himself has blessed. Most experts, including Moore himself, expect Moore’s Law to hold for at least another two decades.

• Technology is not enough.
  — Different architectures have been necessary.
    - Scalar → Super scalar → Vector → Parallel.
Problems?

• With the more advanced architectures, the programming has become more difficult

• Trying to hand optimizing these programs is much like hand optimizing a high-level program
  — It is too complicated for the user because there are too many things to take into account
  — The machine architectures differ so much that an optimization that is effective on one architecture may not be effective on a different architecture
Fortran

• “It was our belief that if Fortran, during its first months, were to translate any reasonable ‘scientific’ source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger.... To this day I believe that our emphasis on the object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like Fortran would have been seriously delayed.

In fact, I believe that we are in a similar, but unrecognized, situation today: in spite of all the fuss that has been made over myriad language details, current conventional languages are still very weak programming aids, and far more powerful languages would be in use today if anyone had found a way to make them run with adequate efficiency.”  John Backus

---

Dependence

• Transformations that are performed have to be safe
  – Produce correct results
• Correct results occur when dependences are satisfied
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

• Pipelined Execution Units
  —Because some instructions take longer to execute

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Equate</th>
<th>Add</th>
<th>Normalize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operands (FO)</td>
<td>Exponents (EE)</td>
<td>Mantissas (AM)</td>
<td>Result (NR)</td>
</tr>
<tr>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4 + c5</td>
</tr>
<tr>
<td>c3</td>
<td>c4</td>
<td>c5</td>
<td>a1</td>
</tr>
</tbody>
</table>

  What is the performance challenge?

• Multiple Execution Units

Adder 1
\[ b_1 + c_1 \]

Adder 2
\[ b_2 + c_2 \]

Adder 3
\[ b_3 + c_3 \]

Adder 4
\[ b_4 + c_4 \]

Results

Vector Operations

• Apply same operation to different positions of one or more arrays
  —Goal: keep pipelines of execution units full
  - Example:
    
    VLOAD V1, A  
    VLOAD V2, B  
    VADD V3, V1, V2  
    VSTORE V3, C  

• How do we specify vector operations in Fortran 77
  
  DO I = 1, 64  
  C(I) = A(I) + B(I)  
  ENDDO  

• Fortran 90
  
  C(1:64) = A(1:64) + B(1:64)
Vector Operations

- Semantics of vector instructions
  - Right hand side operands are fetched before assignment to the left-hand-side
    
    ```
    DO I = 1, 64
    A(I+1) = A(I) + B(I)  vs. A(2:65) = A(1:64) + B(1:65)
    ENDDO
    ```

  - Are they the same?

- Vector operations significantly increase the number of instructions in the machine
  - This can be a problem because it complicates the architecture

Superscalar Machines

- We want to be able to issue multiple instructions per cycle, up to some upper bound

- Superscalars machine accomplish this by having hardware that looks ahead in the instruction stream for operations that are ready to execute
  - These instruction are executed in separate functional units
Superscalar

• For pipelines to be effective, there has to be a large amount of data passed through without an interruption
  — What would cause an interruption?

• Some machines try to fill the execution unit pipelines and produce results at a rate comparable to that of a vector processor
  — Hardware looks ahead in the instruction stream for operations that are ready to execute

• Superscalar processor can continue to issue instructions as long as each instruction it encounters is “ready”

• Some machines can issue instructions out of order
  • What are the performance challenges?

VLIW

• Multiple instructions issue on the same cycle
  — Wide word instruction
  — Usually one instruction slot per functional unit

• What are the performance challenges?
  — Finding enough parallel instructions
  — Avoiding interlocks
    - Scheduling instructions early enough
Processor Parallelism

- While pipelining is an effective way to speed up the execution of a single processor or functional unit, processor parallelism reduces the running time of an application by carrying out different tasks, or the same task on different data sets, simultaneously using multiple processors.

Forms of Processor Parallelism

- Synchronous processor parallelism
  - Replicates whole processors
  - Each processor executes the same program on different portions of the data space
  - Thinking machines, CM-2, and MasPar MP-2
  - Advantages
    - Synchronization is cheap because instructions are executed in lockstep
  - Disadvantage:
    - Not very efficient on branches because they must execute the two sides of a branch in sequence, with different processors disabled on each side.
Forms of Parallel Processors

- **Asynchronous processor parallelism**
  - Each processor executes a different program or different parts of the same program with coarse-grain, explicit synchronization
  - Symmetric multiprocessor (SMPs) made by many different companies
  - Overhead
    - Programs and data must be copied to the different processors
    - Interprocessor synchronization
    - Data needs to be transmitted to other processors that need the result
  - Disadvantage
    - It is essential that parallel execution be used only if there is enough work to compensate for the overhead

SMP Parallelism

- **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

• When is it safe to run two iterations of a loop, $I_1$ and $I_2$ in parallel?
  — If none of the following hold:
    1. $I_1$ writes into a memory location that $I_2$ reads
    2. $I_2$ writes into a memory location that $I_1$ reads
    3. Both $I_1$ and $I_2$ write to the same memory location

• How can we convert this to loop parallelism?
  • Think of loop iterations as tasks
  • In Fortran, the PARALLEL DO statement
    • Each iteration is executed on a separate processor
  • Are the following loops legal?

```
PARALLEL DO I = 1, N
  A(I+1) = A(I) + B(I)
ENDDO
PARALLEL DO I = 1, N
  A(I-1) = A(I) + B(I)
ENDDO
PARALLEL DO I = 1, N
  S = A(I) + B(I)
ENDDO
```

Granularity

• Asynchronous parallel process have significant start-up and synchronization overhead
• To overcome this overhead, the computational task must be fairly large
  — If the task is too small, the overhead dominates
  — If the task is too large, little parallelism is achieved
• What should be parallelized?
  — Outer loops for parallel processors
  — Inner loops for vector processors
Memory Hierarchy

- Problem: the speed of memory is not keeping up with the speed of processors
- Memory is moving farther away from the processor in terms of processor cycles

Memory System Overview

- Measure of performance in memory systems:
  - Latency - time to deliver a single element from memory
  - Bandwidth - number of elements that can be delivered on each cycle
- Ways to deal with processor latency:
  - Avoidance
    - Reduce the typical latencies experienced in a computation
  - This is done by memory hierarchies
  - Tolerance
    - Do something else while waiting for data from memory
    - Prefetching data can help
- Challenge: How can we enhance reuse?
Memory Hierarchy

- 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)
      - C(I) could remain in a register
      - What about A(J)?
    - Strip mine to reuse data from cache - L smaller than cache size
      
      DO JJ = 1, M, L
      DO I = 1, N
      DO J = JJ, JJ+L-1
      A(I) = A(I) + B(J)
      ENDDO
      ENDDO
      ENDDO

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 code
    - 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

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

• Why are the subscripts reversed?
Scalar Machines

- Code makes excellent use of the hardware
- Given a good optimizing compiler, code is generated that comes close to optimal performance possible
  - Need to recognize that \( C(J,1) \) is loop invariant
  - Then the quantity can be put in a register

Problem for Pipelines

- Inner loop of matrix multiply is a reduction

\[
\begin{align*}
A(1,2) \cdot B(2,1) & \to C(1,1) + A(1,1) \cdot B(1,1) \\
A(1,2) \cdot B(2,1) & \to C(2,1) + A(2,1) \cdot B(1,1) \\
A(1,2) \cdot B(2,1) & \to C(3,1) + A(3,1) \cdot B(1,1) \\
A(1,2) \cdot B(2,1) & \to C(4,1) + A(4,1) \cdot B(1,1)
\end{align*}
\]

- Solution:
  - work on several iterations of the \( J \)-loop simultaneously
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
```

Matrix Multiply on Vector Machines

```
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
```
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
    C(J,I) = 0.0
  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
    DO K = 1, N
      DO JJ = 0, 63
        C(JJ,I) = C(JJ,I) + A(JJ,K) * B(K,I)
      ENDDO
    ENDDO
  ENDDO
ENDDO

MatMult for a Vector Machine

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

- Four floating-point multiply-adders and four pipeline stages

```plaintext
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
```

MatMult for a VLIW Machine

```plaintext
DO I = 1, N, 4
    DO J = 1, N, 4
        C(J:J+3,I) = 0.0
        C(J:J+3,I+1) = 0.0
        C(J:J+3,I+2) = 0.0
        C(J:J+3,I+3) = 0.0
        DO K = 1, N
            C(J:J+3,I) = C(J:J+3,I) + A(J:J+3,K) * B(K,I)
            C(J:J+3,I+1) = C(J:J+3,I+1) + A(J:J+3,K) * B(K,I)
                                                (*) (*) (*) (*)
            C(J:J+3,I+2) = C(J:J+3,I+2) + A(J:J+3,K) * B(K,I)
            C(J:J+3,I+3) = C(J:J+3,I+3) + A(J:J+3,K) * B(K,I)
        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
```

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

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

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 iteration over block loops to the inside?
MatMult on a Uniprocessor with Cache

\[
\begin{align*}
\text{DO I = 1, N, S} \\
\text{DO J = 1, N, S} \\
\text{DO p = I, I+S-1} \\
\text{DO q = I, I+S-1} \\
\text{C(q,p) = 0.0} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{DO K = 1, N, T} \\
\text{DO p = I, I+S-1} \\
\text{DO q = J, J+S-1} \\
\text{DO r = K, K+T-1} \\
\text{C(q,p) = C(q,p) + A(q,r) * B(r,p)} \\
\text{ENDDO} \\
\text{ENDDO} \\
\end{align*}
\]

MatMult on a Distributed-Memory MP

\[
\begin{align*}
\text{PARALLEL DO I = 1, N} \\
\text{PARALLEL DO J = 1, N} \\
\text{C(J,I) = 0.0} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{PARALLEL DO I = 1, N, S} \\
\text{PARALLEL DO J = 1, N, S} \\
\text{DO K = 1, N, T} \\
\text{DO p = I, I+S-1} \\
\text{DO q = J, J+S-1} \\
\text{DO r = K, K+T-1} \\
\text{C(q,p) = C(q,p) + A(q,r) * B(r,p)} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO} \\
\end{align*}
\]
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