Program Optimization

CSC 252 Guest Lecture

Chen Ding
Professor

first part based on slides by Randy Bryant and Dave O’Hallaron

Performance Realities

- There’s more to performance than asymptotic complexity
- Constant factors matter too!
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
    - cache performance, parallel performance
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Compilers

- Automatically generating efficient code
  - register allocation
  - instruction scheduling (to improve parallelism)
  - scalar optimization (to remove redundancy)
  - loop optimization/parallelization
- URCS courses
  - CSC 254: Programming Language Design and Implementation
  - CSC 255: Program Analysis and Improvement

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler
- Loop invariant code motion

Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  16 * x \rightarrow x \ll 4
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
Common Subexpression Elimination

- Reuse the result of computed expressions

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

3 multiplications: \( n, (i-1)n, (i+1)n \) 1 multiplication: \( in \)

```asm
leaq 1(%rsi), %rax  # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %rsi     # i*n
imulq %rcx, %rax     # (i+1)*n
imulq %rcx, %r8      # (i-1)*n
addq %rdx, %rsi     # i*n+j
addq %rdx, %rax     # (i+1)*n+j
addq %rdx, %r8      # (i-1)*n+j
imulq %rcx, %rsi  # i*n
addq %rdx, %rsi  # i*n+j
movq %rsi, %rax  # i*n+j
subq %rcx, %rax  # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```c
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- What is the result of `lower("ABC")`?
- What’s the length of string “ABC”?

Optimization Blocker: Procedure Calls

- Why couldn’t compiler move `strlen` outside of inner loop?
  - Unknown procedure may have side effects
  - Alters global state each time called
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`

- Remedies:
  - Use of inline functions
  - GCC does this with `-O2`
  - See web aside ASM:OPT
  - Do your own code motion

```c
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        length++;
        return length;
    }
}
```

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
  - Your way of telling compiler not to check for aliasing
    - An optimizing compiler can handle the cases in the book
      - dependence theory and loop optimization
      - taught in CSC 255
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
- Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can have dramatic performance improvement
- Compilers often cannot make these transformations
- Lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Data Type for Vectors

- Data Types
  - Use different declarations for
    - data_t
      - int
      - float
      - double
- Operations
  - Use different definitions of OP and IDENT
    - + / 0
    - * / 1

- Read in the textbook

Basic Optimizations

```c
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Effect of Basic Optimizations

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Eliminates sources of overhead in loop

Modern CPU Design

Superscalar Processor

- Definition: A superscalar processor can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- Benefit: without programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro
Nehalem CPU

- Multiple instructions can execute in parallel
  1 load, with address computation
  1 store, with address computation
  2 simple integer (one may be branch)
  1 complex integer (multiply/divide)
  1 FP Multiply
  1 FP Add
- Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>11--21</td>
<td>11--21</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>10--23</td>
<td>10--23</td>
</tr>
</tbody>
</table>

Combine4 = Serial Computation (OP = *)

1. Computation (length=8)
   ((((((((d[0] * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7]))
   - Sequential dependence
   - Performance: determined by latency of OP

Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

- Helps integer multiply
  - below latency bound
  - Compiler does clever optimization
- Others don’t improve. Why?
  - Still sequential dependency

\[
x = (x \text{ OP } d[i]) \text{ OP } d[i+1];
\]

x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```c
int i;
for (i = 0; i < limit; i++) {
    x = x OP d[i];
}
```

Method | Integer | Double FP |
-------|---------|-----------|
Operation | Add     | Mult      |
Combine4 | 2.0     | 3.0       |
Unroll 2x | 2.0     | 1.5       |
Latency Bound | 1.0 | 3.0 |

- Perform 2x more useful work per iteration

Loop Unrolling with Reassociation

```c
void unroll2aa_combine(vec_ptr v, data_t *dest) {  
    int length = vec_length(v);  
    int limit = length-1;  
    int i;  
    int local;  
    data_t *d = get_vec_start(v);  
    data_t x = IDENT;  
    for (i = 0; i < limit; i++) {  
        x = x OP d[i];  
    }  
    for (; i < length; i++) {  
        x = x OP d[i];  
    }  
    *dest = x;  
}
```

- Can this change the result of the computation?
Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
- Reason: Breaks sequential dependency

Why is that? (next slide)

Loop Unrolling with Separate Accumulators

```c
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

Reassociated Computation

- What changed:
  - Ops in the next iteration can be started early (no dependency)

- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    - CPE = D/2
  - Measured CPE slightly worse for FP mult

Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- 2x speedup (over unroll2) for Int *, FP +, FP *
- Breaks sequential dependency in a "cleaner," more obvious way

Separate Accumulators

- What changed:
  - Two independent "streams" of operations

- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    - CPE = D/2
  - CPE matches prediction!

Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K

- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially
Achievable Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar Optimum</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code

SIMD: Single Instruction, Multiple Data

- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - with SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results

Using Vector Instructions

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar Optimum</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vector Optimum</td>
<td>0.25</td>
<td>0.53</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput Bound</td>
<td>0.25</td>
<td>0.50</td>
</tr>
</tbody>
</table>

- Make use of SSE Instructions
- Parallel operations on multiple data elements
- See Web Aside OPT:SIMD on CS:APP web page

Getting High Performance

- Good compiler and flags
- Don’t do anything stupid
- Watch out for hidden algorithmic inefficiencies
- Write compiler-friendly code
  - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)
- Tune code for machine
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)

Performance by Parallelism

\[
p = \frac{T_p}{T_p} \times \frac{S_p}{T_p} = \frac{S_p}{T_p} \quad \text{speedup on } p \text{ processors}
\]

\[
\text{Efficiency} = \frac{S_p}{P} \quad \text{work divided among } P \text{ processors}
\]

\[
T_p = \text{time taken on } p \text{ processors}
\]

\[
S_p = \text{speedup on } p \text{ processors}
\]

Typically, \( S_p \leq P \)

Speedup and Efficiency
Amdahl's Law

\[ T_M = \frac{T_T}{\frac{T_T}{T_P} + \frac{(1-x)}{p}} \]

\[ S_P = \frac{T_T}{T_P} \]

\[ S_P = \frac{M}{M + (1-x)M} \]

\[ \frac{1}{x + \frac{1}{p}} \]

Naïve Matrix Multiply

\{implements \( C = C + A*B \)\}

for \( i = 1 \) to \( n \)
for \( j = 1 \) to \( n \)
for \( k = 1 \) to \( n \)
\( C(i,j) = C(i,j) + A(i,k) * B(k,j) \)

Algorithm has \( 2n^3 = O(n^3) \) Flops and operates on \( 3n^2 \) words of memory

\( q \) potentially as large as \( 2n^3 / 3n^2 = O(n) \)

Naïve Matrix Multiply

\{implements \( C = C + A*B \)\}

for \( i = 1 \) to \( n \)
{read row \( i \) of \( A \) into fast memory}
for \( j = 1 \) to \( n \)
{read \( C(i,j) \) into fast memory}
{read column \( j \) of \( B \) into fast memory}
for \( k = 1 \) to \( n \)
\( C(i,j) = C(i,j) + A(i,k) * B(k,j) \)
{write \( C(i,j) \) back to slow memory}

Naïve Matrix Multiply on RS/6000

Number of slow memory references on unblocked matrix multiply

\( m = n^3 \) to read each column of \( B \) \( n \) times
\( + n^2 \) to read each row of \( A \) once
\( + 2n^2 \) to read and write each element of \( C \) once

\( = 2 + 3n^3 \)

\( q \approx \frac{m}{m} = 2n^3 / (n^3 + 3n^2) \)

\( \approx 2 \) for large \( n \), no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row \( i \) of \( A \) times \( B \)
Similar for any other order of 3 loops

Naïve Matrix Multiply on RS/6000

\( T = N^{1.7} \)

12,000 would take 1095 years

Size 2000 took 5 days

\( O(N^3) \) performance would have constant cycles/flop

Performance looks like \( O(N^{1.7}) \)

Slide source: Larry Carter, UCSD
Memory Hierarchy

- Most programs have a high degree of **locality** in their accesses
  - **spatial locality**: accessing things nearby previous accesses
  - **temporal locality**: reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average

<table>
<thead>
<tr>
<th>Speed</th>
<th>1ns</th>
<th>10ns</th>
<th>100ns</th>
<th>10ms</th>
<th>10sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>KB</td>
<td>MB</td>
<td>GB</td>
<td>TB</td>
<td>PB</td>
</tr>
</tbody>
</table>

---

Naïve Matrix Multiply on RS/6000

![Graph showing performance metrics for RS/6000](image)

- TLB miss every iteration
- Cache miss every 16 iterations
- Page miss every 512 iterations

---

Visualizing Matrix Multiplication

**C = A B**

- "stick" of computation is dot product of a row of A with column of B
- \( c_{ij} = \sum a_{ik} \times b_{kj} \)

---

MH algorithm for C = AB

- Partition computation into "cubelets"
  - Each cubelet requires \( s \times s \) submatrix of A and B
  - \( 3 \times s^2 \) data needed; allows \( s^3 \) multiply-adds
- Parent module gives child sequence of cubelets.
  - Choose \( s \) to ensure all data fits into child's memory
  - Child sub-partitions cubelet into still smaller pieces.
- Known as "blocking" or "tiling" long before MH model invented (but rarely applied recursively).

---

Theory of MH algorithm for C = AB

- "Uniform" Memory Hierarchy (UMH) model looks similar to actual computers.
  - Block size, number of blocks per module, and transfer time per item grow by constant factor per level.
  - Naive matrix multiplication is \( O(N^3) \) on UMH.
  - Tiled algorithm is \( O(N^3) \) on UMH.
  - Tiled algorithm gets about 90% "peak performance" on many computers.
  - Moral: good MH algorithm \( \leftrightarrow \) good in practice.
Summary of Performance Optimization

- Read Chapter 5
  - useful techniques carefully explained with examples/exercises
  - loop unrolling / invariant code motion
  - peak processor performance
    - latency/throughput bound
  - instruction-level parallelism
    - reassociation
- Limit of parallelism
  - Amdahl’s law
- Memory hierarchy optimization
  - blocked matrix multiply