Program Optimization

CSci 2021: Machine Architecture and Organization
Lecture #33-35, April 18th-22nd, 2016
Your instructor: Stephen McCamant
Based on slides originally by:
Randy Bryant, Dave O’Hallaron

Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting Instruction-Level Parallelism
- Dealing with Conditionals

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
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
    - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
  - When in doubt, the compiler must be conservative

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler
- Code Motion
  - Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop
  - void set_row(double *a, double *b, long i, long n)
    - long j;
    - for (j = 0; j < n; j++)
      - a[i*n+j] = b[j];
Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

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

- Extracted from CMU 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \rightarrow x \times 4 \]
- Utility machine dependent
- Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Most valuable when it can be done within a loop
  - “Induction variable” has value linear in loop execution count
  - Depends on cost of multiply or divide instruction

```c
for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] = ('A' - 'a');
```

Convert Loop To Goto Form

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

- `strlen` executed every iteration
Calling Strlen

```c
/* My version of strlen */
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    return length;
}
```

- **Strlen performance**: Only way to determine length of string is to scan its entire length, looking for null character.
- **Overall performance, string of length N**:
  - N calls to strlen
  - Require times N, N-1, N-2, ..., 1
  - Overall \(O(N^2)\) performance

Improving Performance

- Move call to strlen outside of loop
- Since result does not change from one iteration to another

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

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2

![Graph showing performance comparison between lower1 and lower2](image)

Optimization Blocker: Procedure Calls

- **Why couldn't compiler move strlen out of inner loop?**
  - Procedure may have side effects
    - Alters global state each time called
    - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure lower could interact with strlen
  - **Warning**: Compiler treats procedure call as a black box
  - Weak optimizations near them

Remedies:

- Use of inline functions
  - GCC does this with -O1
  - Do your own code motion

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

Exercise Break: Weird Pointers

- Can the following function ever return 12, and if so how?
  ```c
  int f(int *p1, int *p2, int *p3) {
      *p1 = 100;
      *p2 = 10;
      *p3 = 1;
      return *p1 + *p2 + *p3;
  }
  ```
  - Yes, for instance:
    ```c
    int a, b;
    f(&a, &b, &a);
    ```

- [https://chimein.cla.umn.edu/course/view/2021](https://chimein.cla.umn.edu/course/view/2021)

Memory Matters

- **Sum rows of n X n matrix a and store in vector b**: /!
  ```c
  void sum_row(double *a, double *b, long n) {
      long i, j;
      for (i = 0; i < n; i++) {
          b[i] = 0;
          for (j = 0; j < n; j++)
              b[i] += a[i*n + j];
      }
  }
  ```
  - Code updates b[i] on every iteration
  - Why couldn't compiler optimize this away?

![Assembly code for sum_row](image)
Memory Aliasing

/// Sum rows of n x n matrix a and store in vector b ///
void sum_rows(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

double A[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};
sum_rows1(A, B, 3);

Value of B:
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]

Removing Aliasing

/// Sum rows of n x n matrix a and store in vector b ///
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}

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 yield 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 structure for vectors */
typed struct{
    size_t len;
    data_t *data;
} vec;

Data Types
- Use different declarations for data_t
  - int
  - long
  - float
  - double

/* retrieve vector element and store at val */
int get_vec_element(vec* v, size_t idx, data_t *val) {
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

Benchmark Computation

void combine1(vec_ptr v, data_t *dest) {
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}

Data Types
- Use different declarations for data_t
  - int
  - long
  - float
  - double

Operations
- Use different definitions of OP and IDENT
  - + / 0
  - * / 1
### Cycles Per Element (CPE)
- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPEn + Overhead
  - CPE is slope of line

### Benchmark Performance
```c
define benchmark
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++)
        *dest = dest OP val;
}
```

#### Method
<table>
<thead>
<tr>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
</tr>
</tbody>
</table>

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

```c
define combine4
{
    long i;
    long 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;
}
```

### Effect of Basic Optimizations
- Eliminates sources of overhead in loop

#### Method
<table>
<thead>
<tr>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
</tr>
</tbody>
</table>

### Modern CPU Design
- Instruction Control
- Functional Units
- Execution
- Address
- Store
- Load
- Arith
- Arith
- Branch
- Instruction Cache
- Fetch
- Control
- Operations
- Retire
- Register File
- Data Cache
- Execution
- Register Updates
- Prediction

### 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 modern CPUs are superscalar.
- Intel: since Pentium (1993)
Pipelined Functional Units

long mult_long(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage \( i \) can start on new computation once values passed to \( i+1 \)
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Haswell CPU
- 8 Total Functional Units
- Multiple instructions can execute in parallel
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- Some instructions take > 1 cycle, but can be pipelined

\[\begin{array}{|c|c|c|c|c|}
\hline
\text{Instruction} & \text{Latency} & \text{Cycles/Issue} \\
\hline
\text{Load/Store} & 4 & 1 \\
\text{Integer Multiply} & 3 & 1 \\
\text{Integer/Long Divide} & 3-30 & 3-30 \\
\text{Single/Double FP Multiply} & 5 & 1 \\
\text{Single/Double FP Add} & 3 & 1 \\
\text{Single/Double FP Divide} & 3-15 & 3-15 \\
\hline
\end{array}\]

x86-64 Compilation of Combine4
- Inner Loop (Case: Integer Multiply)

\[
\text{L519:} = \text{imull}(\%rax,\%rdx,4), \%ecx \\
\text{addq } \$1, \%rdx \\
\text{cmpq } \%rdx, \%rbp \\
\text{jl } .L519
\]

Method | Integer | Double FP
--- | --- | ---
Operation | Add | Mul | Add | Mul
Combine4 | 1.27 | 3.01 | 3.01 | 5.01
Latency Bound | 1.00 | 3.00 | 3.00 | 5.00

Effect of Loop Unrolling
- Helps integer add
- Achieves latency bound
- Others don’t improve. Why?
- Still sequential dependency

Loop Unrolling (2x1)

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

Method | Integer | Double FP
--- | --- | ---
Operation | Add | Mul | Add | Mul
Combine4 | 1.27 | 3.01 | 3.01 | 5.01
Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01
Latency Bound | 1.00 | 3.00 | 3.00 | 5.00
Loop Unrolling with Reassociation (2x1a)

```c
void unroll2aa_combine(data_t *dest) {
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Can this change the result of the computation?
Yes, for FP. Why?

Reassociated Computation

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

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

**Overall Performance**
- N elements, D cycles latency/operation
- \((N/2+1)D\) cycles:
  - \(CPE = D/2\)

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>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Int + makes use of two load units
  - \(x0 = x0 \text{ OP} \ d[i]\)
  - \(x1 = x1 \text{ OP} \ d[i+1]\)
- 2x speedup (over unroll2) for Int *, FP +, FP *

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>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1</td>
<td>1.01</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x1a</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x2</td>
<td>0.81</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency
    - \(x = x \text{ OP} \ (d[i] \text{ OP} \ d[i+1])\)
  - Why is that? (next slide)

- Different form of reassociation

Separate Accumulators

\[ x0 = x0 \text{ OP} \ d[i]; \]
\[ x1 = x1 \text{ OP} \ d[i+1]; \]

**What changed:**
- Two independent “streams” of operations

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

What Now?
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

Unrolling & Accumulating: Double *

**Case**
- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

<table>
<thead>
<tr>
<th>( K )</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
</tr>
<tr>
<td>4</td>
<td>1.25, 1.26</td>
</tr>
<tr>
<td>6</td>
<td>0.84, 0.88</td>
</tr>
<tr>
<td>8</td>
<td>0.63</td>
</tr>
<tr>
<td>10</td>
<td>0.51</td>
</tr>
<tr>
<td>12</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Accumulators

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>Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

Programming with AVX2

**YMM Registers**
- 16 total, each 32 bytes
- 32 single-byte integers
- 16 16-bit integers
- 8 32-bit integers
- 8 single-precision floats
- 4 double-precision floats
- 1 single-precision float
- 1 double-precision float

SIMD Operations

**SIMD Operations: Single Precision**

\[ \text{vaddsd} \ %ymm0, \ %ymm1, \ %ymm2 \]

**SIMD Operations: Double Precision**

\[ \text{vaddpd} \ %ymm0, \ %ymm1, \ %ymm2 \]
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 Best</td>
<td>0.54</td>
<td>1.01</td>
</tr>
<tr>
<td>Vector Best</td>
<td>0.06</td>
<td>0.24</td>
</tr>
<tr>
<td>Latency Bound</td>
<td>0.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput Bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec Throughput</td>
<td>0.06</td>
<td>0.12</td>
</tr>
</tbody>
</table>

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

What About Branches?

- Challenge
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy
  - When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design

- Branch Outcomes
  - When encounter conditional branch, cannot determine where to continue fetching
    - Branch Taken: Transfer control to branch target
    - Branch Not-Taken: Continue with next instruction in sequence
    - Cannot resolve until outcome determined by branch/integer unit

- Branch Prediction
  - Idea
    - Guess which way branch will go
    - Begin executing instructions at predicted position
    - But don’t actually modify register or memory data
  - Branch Prediction Through Loop
    - Assume vector length = 100
    - Predict Taken (OK)
    - Predict Taken (Oops)
    - Read invalid location

What About Branches?

- Challenge
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy
  - When encounters conditional branch, cannot reliably determine where to continue fetching
### Branch Misprediction Invalidation

- Assume vector length = 100
- Predict Taken (OK)
- Predict Taken (Oops)
- Invalidate

```asm
401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp trax,%rdx
401034: jne 401029
```

### Branch Misprediction Recovery

- Definitely not taken
- Reload Pipeline

```asm
401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add $0x8,%rdx
401031: cmp trax,%rdx
401034: jne 401029
```

### 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)