ECE 454
Computer Systems Programming

Program Optimizations

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
Content

- History and overview of compilers
- Basic compiler optimizations
- Program optimizations
- Advanced optimizations
  - Parallel unrolling
  - Profile-directed feedback
Program Optimization

Example: Vector Sum Function
Vector Procedures

- **vec_ptr new_vec(int len)**
  - Create vector of specified length
- **int get_vec_element(vec_ptr v, int index, int *dest)**
  - Retrieve vector element at index, store at *dest
  - Return 0 if out of bounds, 1 if successful
- **int *get_vec_start(vec_ptr v)**
  - Return pointer to start of vector data
Vector Sum Function: Original Code

```c
void vsum(vec_ptr v, int *dest) {
    int i;
    *dest = 0;
    for (i = 0; i < vec_len(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

- Vector sum: add all the vector elements together
  - Store the resulting sum at destination location *dest
- Performance Metric: CPU Cycles per Element (CPE)
  - CPE = 42.06 (Compiled -g)
After Compiler Optimization

void vsum1(vec_ptr v, int *dest) {
    int i;
    *dest = 0;
    for (i = 0; i < vec_len(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}

- Impact of compiler optimization
  - vsum: CPE = 42.06 (compiled -g)
  - vsum1: CPE = 31.25 (compiled -O2)
    - Will compile with –O2 from now on
  - Improvements due to better scheduling and register allocation
  - However, lots of missed opportunities!

a C-code view of the resulting assembly instructions, i.e., vsum1 is the same as vsum.
Optimization Blocker: Procedure Calls

• **Why didn’t compiler move vec_len out of the loop?**
  • Procedure vec_len may have side effects
    • Changes global state
  • Function may not be deterministic
    • Reads/depends on global state

• **Why doesn’t compiler look at code for vec_len?**
  • Linker may overload with different version, unless declared static
  • Interprocedural optimization is not used extensively due to cost

• **Be careful**
  • Compiler treats procedure call as a black box
  • Limited optimizations in and around them
Manual LICM

- CPE = 31.25 ➔ 20.66
- Next opportunity: Inlining get_vec_element()?
  - Compiler may not have thought it was worth it
  - -O2 avoids increasing code size too much

```c
void vsum1(vec_ptr v, int *dest) {
    int i;
    *dest = 0;
    for (i=0; i<vec_len(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}

void vsum2(vec_ptr v, int *dest) {
    int i;
    int length = vec_len(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```
Manual Inlining

- Inlining: replace a function call with its body
  - Shouldn’t normally have to do this manually
- Inlining enables further optimizations

void vsum2(vec_ptr v, int *dest) {
    int i;
    int length = vec_len(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}

void vsum3i(vec_ptr v, int *dest) {
    int i;
    int length = vec_len(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        int *data = get_vec_start(v);
        val = data[i];
        *dest += val;
    }
}
Further LICM, val Unnecessary

- CPE = 20.66 ➔ 6.00
  - Compiler may not always inline when it should
  - It may not know how important a certain loop is
  - It may be blocked by a missing function definition

```c
void vsum3i(vec_ptr v, int *dest)
{
    int i;
    int length = vec_len(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        int *data = get_vec_start(v);
        val = data[i];
        *dest += val;
    }
}

void vsum3(vec_ptr v, int *dest)
{
    int i;
    int length = vec_len(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest += data[i];
    }
}
```
Reduce Unnecessary Memory Refs

- CPE = 6.00 ➞ 2.00

- Don’t need to store in `dest` until the end
  - Can use local variable `sum` instead
  - It is more likely to be allocated in a register
  - Avoids 1 memory read, 1 memory write per iteration

```c
void vsum3(vec_ptr v, int *dest)
{
    int i;
    int length = vec_len(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest += data[i];
    }
}

void vsum4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_len(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
    {
        sum += data[i];
        *dest = sum;
    }
}
```
Why are Pointers so Hard for Compilers?

- In previous slide, it is difficult for the compiler to promote a pointer dereference to a register, what makes it hard?
  - Compiler needs to know that no other pointer can point to *dest
  - Otherwise that pointer could be used to change *dest

- Pointer aliasing
  - Two different pointers might point to a single location

```c
int x = 5, y = 10;
int *dest = &x;
int *rand = &y;
if (one_in_a_million){
    rand = dest;
}
```

// compiler must assume value of rand may be the same as the value of dest, i.e., *rand may equal x from here on, though unlikely.
Minimizing the Impact of Pointers

• Easy to over-use pointers in C/C++
  • Benefit: direct access to storage structures
  • Drawback: since pointers allow doing address arithmetic, pointer analysis becomes even harder

• Get in the habit of introducing local variables
  • E.g., when accumulating within loops
  • Your way of telling the compiler to not worry about aliasing
Understanding Instruction-Level Parallelism:

Unrolling and Software Pipelining
Superscalar CPU Design

Instruction Control
- Instruction Cache
- Instruction Decode
- Fetch Control
- Retirement Unit
- Register File

Execution
- Functional Units
  - Integer/Branch
  - General Integer
  - FP Add
  - FP Mult/Div
  - Load
  - Store

Data Cache

Fetch Control
- Address
- Instrs.
- Operations

Prediction OK?
- Register Updates

Retirement Unit
- Addr
- Data
Assumed CPU Capabilities

- Multiple instructions can execute in parallel
  - 1 load
  - 1 store
  - 2 integer (one may be branch)
  - 1 FP addition
  - 1 FP multiplication or division
Assumed CPU Capabilities

- Several instructions take > 1 cycle, but can be pipelined
  - Instruction | Latency | Cycles/Issue
  - Load / Store | 3 | 1
  - Integer Multiply | 4 | 1
  - Double/Single FP Multiply | 5 | 2
  - Double/Single FP Add | 3 | 1
  - Integer Divide | 36 | 36
  - Double/Single FP Divide | 38 | 38
## Intel Core i7

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer</th>
<th>Single-precision</th>
<th>Double-precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Latency</td>
<td>Issue</td>
</tr>
<tr>
<td>Addition</td>
<td>1</td>
<td>3</td>
<td>0.33</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Division</td>
<td>11–21</td>
<td>10–15</td>
<td>5–13</td>
</tr>
</tbody>
</table>
void vsum4(vec_ptr v, int *dest)
{
    int i;                          // i      => %edx
    int length = vec_len(v);       // length => %esi
    int *data = get_vec_start(v);  // data   => %eax
    int sum = 0;                    // sum    => %ecx
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}

.L24:
    addl (%eax,%edx,4),%ecx        # sum += data[i]  // data + (i*4)
    incl %edx                      # i++
    cmpl %esi,%edx                 # compare i and length
    jl  .L24                      # if i < length goto L24
addl includes a load operation!

addl (%eax,%edx,4),%ecx

3 cycles
%edx.0 (i)
load data[i]

1 cycle
%ecx.0 (sum)
t.1
addl

%ecx.1 (sum)
Visualizing the vsum4 Loop

.L24:
  addl (%eax,%edx,4),%ecx  
  incl %edx  
  cmpl %esi,%edx  
  jl .L24

# Loop code only:
# addl (%eax,%edx,4),%ecx  
# incl %edx  
# cmpl %esi,%edx  
# jl .L24

• Height of operation denotes latency
• Operation cannot begin until operands are available
4 Iterations of vsum4

- Unlimited resource analysis
  - Assume operation can start as soon as operands available
  - Operations for multiple iterations overlap in time

- CPE = 1.0, but need to issue 4 integer ops per cycle

- Performance on assumed CPU
  - CPE = 2.0, number of parallel integer ops (2) is the bottleneck
Loop Unrolling 3 Times: \texttt{vsum5}

- Combine multiple iterations into single loop body
- Amortizes loop overhead across multiple iterations
- Finish extras at end

- Measured CPE = 1.33

\begin{verbatim}
void vsum4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_len(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}

void vsum5(vec_ptr v, int *dest)
{
    int i;
    int length = vec_len(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < limit; i+=3) {
        sum += data[i];
        sum += data[i+1];
        sum += data[i+2];
    }
    // fix up
    for ( ;i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
\end{verbatim}
Visualizing Unrolled Loop: vsum5

Loads can pipeline, since they don’t have dependences
Executing with Loop Unrolling: vsum5

- Predicted Performance
  - Can complete iteration in 3 cycles
  - Should give CPE of 1.0
Why Unroll 3 Times?

Unroll 2 times: what is the problem?
Vector multiply function: vprod

```c
void vprod4(vec_ptr v, int *dest) {
    int i;
    int length = vec_len(v);
    int *data = get_vec_start(v);
    int x = 1;
    for (i = 0; i < length; i++)
        x *= data[i];
    *dest = x;
}
```

.L24:
    # Loop code only:
    imull (%eax,%edx,4),%ecx   # x *= data[i]   // data+i*4
    incl %edx                   # i++
    cmpl %esi,%edx             # compare i and length
    jl .L24                    # if i < length goto L24

```
```
3 Iterations of vprod

With unlimited resource analysis, limiting factor becomes latency of integer multiplier, CPE = 4.0
Unrolling: Performance Summary

<table>
<thead>
<tr>
<th>Unrolling Degree</th>
<th>Relevant FU Latency</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsum</td>
<td>1 cycle</td>
<td>2.00</td>
<td>1.50</td>
<td>1.33</td>
<td>1.50</td>
<td>1.25</td>
<td>1.06</td>
</tr>
<tr>
<td>vprod</td>
<td>4 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.00</td>
</tr>
<tr>
<td>vFPsum</td>
<td>3 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.00</td>
</tr>
<tr>
<td>vFPprod</td>
<td>5 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.00</td>
</tr>
</tbody>
</table>

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations
- Can we do better for vprod? What is the current performance limitation?
Visualizing vprod

Computation:

\[((\ldots(1 \times x_0) \times x_1) \times x_2) \times x_3) \times x_4) \times x_5) \times x_6) \times x_7)\ldots\]

- Performance
  - $x_0, x_1 \ldots$ are data[0], data[1]…
  - N elements, D cycles/operation
  - $D = \text{latency of imull}$
  - $N \times D \text{ cycles} = N \times 4$
Parallel Unrolling 1: vprod5pu1

• Assume unlimited hardware resources from now on

• Optimization
  • Multiply pairs of elements
  • Then update product
  • Finish extras at end

• Performance
  • CPE = 2

```c
void vprod5pu1(vec_ptr v,
               int *dest)
{
    int length = vec_len(v);
    int limit = length - 1;
    int *data = get_vec_start(v);
    int x = 1; int i;

    for (i = 0; i < limit; i+=2) {
        x = x * (data[i] * data[i+1]);
    }
    if (i < length) // fix-up
        x *= data[i];
    *dest = x;
}
```
2 Iterations of vprod5pu1

tmp = data[i] * data[i+1]

Still have the data dependency between multiplications, but it’s one dependency every two elements
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * data[i]) * data[i+1];
}

All multiplies performed in sequence
CPE = 4.00

/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = x * (data[i] * data[i+1]);
}

Multiplies across difference iterations can overlap, CPE = 2
Parallel Unrolling 2: vprod5pu2

- **Optimization**
  - Accumulate in two different products
  - Can be performed simultaneously
  - Combine at end

- **Performance**
  - CPE = 2

```c
void vprod6pu2(vec_ptr v, int *dest)
{
    int length = vec_len(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    int i;
    for (i = 0; i < limit; i+=2)
        x0 *= data[i];
        x1 *= data[i+1];
    }
    if (i < length) // fix-up
        x0 *= data[i];
    *dest = x0 * x1;
}
```
for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
}
...
*dest = x0 * x1;
Trade-offs With Loop Unrolling

- **Benefits**
  - Reduces loop operations
    - E.g., condition check & loop variable increment reduced
    - Reason for integer add (vsum5) to achieve a CPE of 1
  - Improves parallelism by reducing data dependencies
    - However, often requires manually rewriting loop body

- **Drawbacks**
  - Increased code size
  - Register pressure, several registers needed to hold sums/products
    - IA32: Only 6 usable integer registers, 8 FP registers
    - When not enough registers, must spill temporaries onto stack
      - Wipes out any performance gains!
Effective Parallel Unrolling

• Algorithmic/mathematical requirement
  • Operation being combined must be associative & commutative
    • OK for integer multiplication
    • Not strictly true for floating point operations
      • OK for most applications

• Hardware requirement
  • Pipelined functional units, superscalar execution
  • Lots of registers to hold sums/products
Summary
Summary of Optimizations

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization</th>
<th>Applied to</th>
<th>Manual or GCC</th>
<th>CPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsum</td>
<td>-g</td>
<td></td>
<td></td>
<td>42.06</td>
</tr>
<tr>
<td>vsum1</td>
<td>-O2</td>
<td></td>
<td>GCC</td>
<td>31.25</td>
</tr>
<tr>
<td>vsum2</td>
<td>LICM</td>
<td>vec_len()</td>
<td>manual</td>
<td>20.66</td>
</tr>
<tr>
<td>vsum3</td>
<td>Inlining + LICM</td>
<td>get_vec_element()</td>
<td>manual</td>
<td>6.00</td>
</tr>
<tr>
<td>vsum4</td>
<td>mem-ref reduction</td>
<td>*dest</td>
<td>manual</td>
<td>2.00</td>
</tr>
<tr>
<td>vsum5</td>
<td>unrolling (3x)</td>
<td>for loop</td>
<td>-funroll-loops</td>
<td>1.33</td>
</tr>
<tr>
<td>vprod4</td>
<td>(same as vsum4)</td>
<td></td>
<td></td>
<td>4.00</td>
</tr>
<tr>
<td>vprod5pu1</td>
<td>par. unrolling 1</td>
<td>for loop</td>
<td>manual</td>
<td>2.0</td>
</tr>
<tr>
<td>vprod5pu2</td>
<td>par. unrolling 2</td>
<td>for loop</td>
<td>manual</td>
<td>2.0</td>
</tr>
</tbody>
</table>
Takeaways

• Before you start: will optimization be worth the effort?
  • Profile to estimate potential benefits

• Get the most out of your compiler before going manual
  • Trade-off: manual optimization vs readable/maintainable code

• Exploit compiler optimizations for maintainable code
  • Use lots of functions and write modular code
  • Enable debugging/tracing code (enabled by static flags)
Takeaways

• Limit use of pointers
  • Reduce pointer-based arrays and pointer arithmetic
  • Function pointers and virtual functions (unfortunately)

• For highly performance-critical code:
  • Look at assembly for optimization opportunities
  • Consider the instruction-parallelism capabilities of CPU
How to know what the compiler has done?

- run: `objdump –d a.out`
  - prints a listing of instructions, with instruction address, encoding, and assembly

```c
void main() {
    int i = 5;
    int x = 10;
    ...
}
```

<table>
<thead>
<tr>
<th>instruction address</th>
<th>instruction</th>
<th>encoding</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048390:</td>
<td>&lt;main&gt;:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048394:</td>
<td>55</td>
<td></td>
<td>push %ebp</td>
</tr>
<tr>
<td>8048395:</td>
<td>89 e5</td>
<td></td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>8048397:</td>
<td>83 ec 10</td>
<td></td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td>804839a:</td>
<td>c7 45 f8 05 00 00 00</td>
<td></td>
<td>movl $0x5,-0x8(%ebp)</td>
</tr>
<tr>
<td>80483a1:</td>
<td>c7 45 fc 0a 00 00 00</td>
<td></td>
<td>movl $0xa,-0x4(%ebp)</td>
</tr>
</tbody>
</table>
Advanced Topics

• Profile-Directed Feedback (PDF)
  • Provide gprof-like measurements as input to compiler
  • Allows compiler to make smarter choices/optimizations
    • Eg., handle the “if (one-in-a-million)” case well

• Just-in-Time compilation and optimization (JIT)
  • Performs a simple version of many of the basic compiler optimizations at runtime, exploits online PDF
  • E.g., Java JIT

• Current hot topics in compilation:
  • Automatic parallelization (exploiting multicores)